SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 20(9), 899–928 (SEPTEMBER 1990)

Calendrical Calculations

NACHUM DERSHOWITZ AND EDWARD M. REINGOLD
Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801-2987, U.S.A.

SUMMARY

A unified, algorithmic presentation is given for the Gregorian (current civil), ISO, Julian (old civil), Islamic (Moslem), and Hebrew (Jewish) calendars. Easy conversion among these calendars is a byproduct of the approach, as is the determination of secular and religious holidays.

KEY WORDS Calendar Holidays Gregorian calendar Hebrew calendar Islamic calendar ISO calendar Julian calendar

Teach us to number our days, that we may attain a wise heart.
—Psalms 90:12

INTRODUCTION

Calendrical calculations are ubiquitous. Banks need to calculate interest on a daily basis. Operating systems need to switch to and from daylight savings time. Dates of secular and religious holidays need to be computed for consideration in planning events. Paychecks need to be issued on weekly, biweekly, or monthly schedules. Bills and statements must be generated periodically. Most of these calculations are not difficult because the rules of our civil calendar (the Gregorian calendar) are straightforward.

Complications begin when we need to know the day of the week on which a given date falls or when various religious holidays based on other calendars occur. These complications lead to difficult programming tasks—not difficult in an algorithmic sense, but difficult because it can be extremely tedious to delve, for example, into the complexities of the Hebrew calendar and its relation to the civil calendar.

The purpose of this paper is to present, in a completely algorithmic form, a description of five basic calendars and how they relate to one another: the present civil calendar (Gregorian), the recent ISO commercial calendar, the old civil calendar (Julian), the Islamic (Moslem) calendar, and the Hebrew (Jewish) calendar. Information that is sufficiently detailed to allow computer implementation is difficult to find for the Islamic and Hebrew calendars since the published material is often inaccessible, ecclesiastically oriented, incomplete, inaccurate, based on extensive tables, overburdened with extraneous material, focused on shortcuts for hand calculation to

0038–0644/90/090899–30$15.00
© 1990 by John Wiley & Sons, Ltd.
Received 31 August 1989
Revised 27 March 1990

avoid complicated arithmetic or to check results, or difficult to find in English. Most existing computer programs are proprietary, incomplete, or inaccurate.

The need for such a secular presentation in the public domain was made clear to us when the second author, in implementing a calendar/diary feature for GNU Emacs,[1] found difficulty in gathering and interpreting appropriate source materials that describe the interrelationships among the various calendars and the determination of the dates of holidays. The material presented in this paper, in the form of COMMON LISP[2] functions, unifies the calculations for all five calendars.* We have chosen Lisp as the vehicle for implementation because it encourages functional programming and has a trivial syntax, nearly self-evident semantics, historical durability, and wide distribution.

It is not the intention of this paper to give a detailed historical treatment of the material, nor, for that matter, a mathematical one; our goal is to give a computational treatment that will prove useful to programmers. Thus, although we give some necessary historical, religious, mathematical, and astronomical details in the text, the focus of the presentation is in the Lisp functions. Full historical/religious details and mathematical/astronomical underpinnings of the calendars can be pursued in the references. We have chosen not to optimize the code at the expense of algorithmic clarity; consequently, considerable improvements in economy are possible (some possibilities are pointed out).

In the next section we describe the underlying unifying idea of all the calculations. The details of specific calendars are presented in subsequent sections. Historically, the oldest of the calendars that we consider is the Julian (the roots of which date back to the ancient Roman empire). Next oldest is the Hebrew calendar (fourth century), followed by the Islamic calendar (seventh century), followed by the Gregorian modification to the Julian calendar (sixteenth century). Finally, the International Organization for Standardization's ISO calendar is of twentieth century origin. For expository purposes, however, we present the Gregorian calendar first because it is the most popular calendar currently in use; then we give the ISO calendar which depends wholly on the Gregorian. Since the Julian calendar is so close in substance to the Gregorian, we present it next. Then we give the Islamic calendar which, because of its simplicity, is easy to implement. Finally, we present the Hebrew calendar, the most complicated of the five calendars, and the most difficult to implement. In the penultimate section, we give algorithms for calculating the dates of all major and many minor secular and religious holidays. The final section contains descriptions of two other calendars for which we do not provide algorithmic details.

ABSOLUTE DAY NUMBERS

Over the centuries, human beings have devised an enormous variety of methods for specifying dates.[3–6] (An exceptional survey can be found in the *Encyclopædia of Religion and Ethics*,[7] vol. III, pp. 61–141 and vol. V, pp. 835–894.) None are ideal

*To insure correctness, all code in this paper was typeset directly from working Lisp functions. We will gladly provide these Lisp functions in electronic form: send an empty electronic mail message to `reingold@cs.uiuc.edu` with the subject line containing precisely the phrase 'send-cal'; your message will be answered automatically.

computationally, however, because all have idiosyncrasies resulting from attempts to coordinate a convenient human labeling with lunar and/or solar phenomena. All of the calendars that we consider have an integral number of days in a month and an integral number of months in a year, but the astronomical events with which they are supposed to correlate do not follow such a convenient pattern, nor are the precise lengths of astronomical cycles constant over time. Rather, the *mean* length of a (synodic) month is currently 29.5306 (mean) days and the current *mean* length of a (tropical) year is 365.2422 days. The different calendars differ in the accuracy with which their months and years approximate these figures.

For a computer implementation, the easiest way to reckon time is simply to count days: Establish an arbitrary starting point as day 1 and specify a date by giving a day number relative to that starting point;[8] a single thirty-two bit integer allows the representation of more than 11.7 million years. Such a reckoning of time is, evidently, extremely awkward for human beings and is not in common use, except among astronomers who use *Julian day numbers* to specify dates.[9]

We have chosen Monday, January 1, 1 C.E.* (Gregorian) as our absolute date 1 and count forward day-by-day from there. Of course, this is anachronistic because there was no year 1 C.E. on the Gregorian calendar—the Gregorian calendar was devised only in the sixteenth century—so by January 1, 1 C.E. (Gregorian) we mean the day we get if we extrapolate backwards from the present; this day turns out to be Monday, January 3, 1 C.E. (Julian).

We should thus think of the passage of time as a sequence of days numbered 1, 2, 3, ... that the various human-oriented calendars label differently. For example, absolute day 710347 is called November 12, 1945 C.E. on the Gregorian calendar, day 1 of week 46 of 1945 C.E. on the ISO calendar, October 30, 1945 C.E. on the Julian calendar, Dhu al-Hijjah 6, 1364 A.H.† on the Islamic calendar, and Kislev 7, 5706 A.M.‡ on the Hebrew calendar. All that is required for calendrical conversion is to be able to convert to and from this absolute calendar. We give, in subsequent sections, Lisp functions to do the conversions for the Gregorian, ISO, Julian, Islamic, and Hebrew calendars. The algorithms given in this paper do *not* generally work for non-positive absolute dates.

The date Monday, January 1, 1 C.E. (Gregorian), though arbitrarily chosen as our starting point, has two desirable characteristics. First, it is early enough that almost all dates of interest are represented by positive integers; in any case, using any earlier date would be problematic because of historical irregularities in the application of the Julian leap year rule. Second, since the day is a Monday, determining the day of the week amounts to taking the absolute date modulo seven—zero is Sunday, one is Monday, and so forth.

LISP PRELIMINARIES

For readers unfamiliar with Lisp, this section provides the bare necessities. Other details will be mentioned in passing as they are used; a complete description can be found in COMMON LISP: *The Language*.[2]

---

*Common era; or, A.D.

†*Anno hegirae*; in the year of the Hegira (Mohammed's flight to Medina).

‡*Anno mundi*; in the (traditional) year of the world (since creation).

All functions in Lisp are written in prefix notation: If `f` is a defined function, then

```
(f e0 e1 e2 ... en)
```

applies `f` to the $n+1$ arguments `e0`, `e1`, `e2`, ..., `en`. Thus

```
(+ 1 -2 3)
```

adds the three numbers and returns the value 2;

```
(<= 1 2 3)
```

checks that the three numbers are in nondecreasing order and yields true (`t` in Lisp).

Lists are Lisp's main data structure. To construct a list `(e0 e1 e2 ... en)` the expression

```
(list e0 e1 e2 ... en)
```

is used. The function `(nth i l)` extracts the ith element of the list `l`, indexing from zero; the predicate `(member x l)` tests if `x` is an element of `l`. To get the first (indexed zero), second, or third elements of a list, we use the functions `first`, `second`, and `third`, respectively. The empty list is represented by `nil`.

Functions are defined using the `defun` command, which has the following syntax:

```
(defun function-name (param1 ... paramn)
   expression)
```

For example, we can define a function (unavailable in COMMON LISP) to return the (truncated) integer quotient of two integers:

```
(defun quotient (m n)
   (floor (/ m n)))
```

We will represent all dates on the Gregorian, Julian, Islamic, and Hebrew calendars by a list of the form (*month day year*) in which *month*, *day*, and *year* are each integers. (COMMON LISP places no *a priori* upper bound on the size of integers; none of our calculations require more than 32-bit integers for dates in the next twenty thousand years; 24 bits suffice for all of the calculations, except as noted.) To extract the individual components of such a date we use the following access functions:

```
(defun extract-month (date)
;; Month field of date = (month day year).
  (first date))

(defun extract-day (date)
;; Day field of date = (month day year).
  (second date))

(defun extract-year (date)
;; Year field of date = (month day year).
```

```
(third date))
```

Notice that the double semicolons demarcate comments.

For convenience in expressing our calendar functions in Lisp, we introduce a macro to compute sums. The expression

```
(sum f i k p)
```

computes $\sum_{i \geq k, p(i)} f(i)$; that is, the expression $f(i)$ is summed for all $i = k, k+1, \ldots$, continuing only as long as the condition $p(i)$ holds. The (opaque) COMMON LISP definition of sum is as follows:

```
(defmacro sum (expression index initial condition)
;; Sum expression for index = initial and successive integers,
;; as long as condition holds.
  (let* ((temp (gensym)))
    `(do ((,temp 0 (+ ,temp ,expression))
          (,index ,initial (1+ ,index)))
         ((not ,condition) ,temp))))
```

THE GREGORIAN CALENDAR

The calendar in use today in most countries is the new style, or Gregorian, calendar designed by a commission assembled by Pope Gregory XIII in the sixteenth century.[3, 5, 6, 10–13] This strictly solar calendar is based on a 365-day common year divided into twelve months of lengths 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, and 31 days, and on 366 days in leap years, the extra day being added to make the second month 29 days long:

(1) January	31 days	(7) July	31 days
(2) February	28 {29} days	(8) August	31 days
(3) March	31 days	(9) September	30 days
(4) April	30 days	(10) October	31 days
(5) May	31 days	(11) November	30 days
(6) June	30 days	(12) December	31 days

The leap-year structure is given in curly brackets—a year is a leap year if it is divisible by 4 and is not a century year (multiple of 100) or if it is divisible by 400. For example, 1900 was not a leap year, while 2000 will be. The Gregorian calendar differs from its predecessor, the old style or Julian calendar, only in that the Julian calendar did not include the century rule for leap years—all century years were leap years.

The Julian calendar was instituted in 45 B.C.E.* by Julius Cæsar on January 1, 709 A.U.C.†; it was a modification of an ancient Egyptian calendar. Since every fourth

*Before the common era; or, B.C.

†*Ab urbe condita*; from the (traditional) founding of the city (of Rome).

year was a leap year, a cycle of 4 years contained $4 \times 365 + 1 = 1461$ days, giving an average length of year of 365.25 days. This is somewhat more than the mean length of the solar year, and over the centuries the calendar slipped with respect to the solar year. By the sixteenth century, the date of the vernal (spring) equinox had shifted from around March 21 to around March 11. If this error were not corrected, eventually Easter, the date of which depends on the vernal equinox, would migrate through the whole calendar year. Pope Gregory instituted only a minor change in the calendar—century years not divisible by 400 would no longer be leap years. Thus, three out of four century years are common years, giving a cycle of 400 years containing $400 \times 365 + 97 = 146097$ days and an average year length of $146097/400 = 365.2425$ days. He also corrected the accumulated 10-day error in the calendar by proclaiming that Thursday, October 4, 1582 C.E., the last date in the old style (Julian calendar), would be followed by Friday, October 15, 1582 C.E., the first day of the new style (Gregorian) calendar. Catholic countries followed his rule, but Protestant countries resisted: Spain, Portugal, and Italy adopted it immediately, as did the Catholic states in Germany. The Protestant parts of Germany waited until 1700 to adopt it, Great Britain and its colonies (including the United States) waited until 1752, Russia held out until after the revolution in 1918, and Bulgaria until 1920 (an extensive list of dates of adoption of the Gregorian calendar can be found in the *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*[9]).

To convert from a Gregorian date to an absolute date, we first need a function that gives the last day (that is, the number of days) for any Gregorian month. This is easily written:

```
(defun last-day-of-gregorian-month (month year)
;; Last day in Gregorian month during year.
  (if ;; February in a leap year
      (and (= month 2)
           (= (mod year 4) 0)
           (not (member (mod year 400) (list 100 200 300))))
;; Then return
      29
;; Else return
    (nth (1- month)
         (list 31 28 31 30 31 30 31 31 30 31 30 31))))
```

The function `if` has three arguments, a Boolean condition, a then-expression, and an else-expression. The function `1-` decrements an integer by one (the similar function `1+` increments by one). Here, and elsewhere, arrays would be more efficient than lists.

The calculation of the absolute date from the Gregorian date (which has been described[8] as 'impractical') can now be done by counting the number of days in prior years—both common and leap years, the number of days in prior months of the current year, and the number of days in the current month:

```
(defun absolute-from-gregorian (date)
;; Absolute date equivalent to the Gregorian date.
  (let* ((month (extract-month date))
```

```
        (year (extract-year date)))
;;  Return
    (+ (extract-day date)     ;; Days so far this month.
       (sum                   ;; Days in prior months this year.
        (last-day-of-gregorian-month m year) m 1 (< m month))
       (* 365 (1- year))      ;; Days in prior years.
       (quotient (1- year) 4);; Julian leap days in prior years...
       (-                     ;; ...minus prior century years...
        (quotient (1- year) 100))
       (quotient              ;; ...plus prior years divisible...
        (1- year) 400))))     ;; ...by 400.
```

The COMMON LISP construct `let*` defines a sequence of constants (possibly in terms of previously defined constants) and ends with an expression the value of which is returned by the construct.

Calculating the Gregorian date from the absolute date d involves sequentially determining the year, month, and day of the month. The year is first closely approximated from below by $\lfloor d/366 \rfloor$ and then found precisely by stepping through subsequent years (that is, by a linear search). The month is then found by a similar linear process, and the day of the month is determined by subtraction:

```
(defun gregorian-from-absolute (date)
;; Gregorian (month day year) corresponding absolute date.
  (let* ((approx (quotient date 366));; Approximation from below.
         (year          ;; Search forward from the approximation.
          (+ approx
             (sum 1 y approx
                  (>= date
                      (absolute-from-gregorian
                       (list 1 1 (1+ y)))))))
         (month          ;; Search forward from January.
          (1+ (sum 1 m 1
                   (> date
                      (absolute-from-gregorian
                       (list m
                             (last-day-of-gregorian-month m year)
                             year))))))
         (day            ;; Calculate the day by subtraction.
          (- date (1- (absolute-from-gregorian
                       (list month 1 year))))))
    ;;  Return
    (list month day year)))
```

It is not hard to determine better approximations for the year, and the same is true for the approximations we will use for other calendars. In fact, the year can be

determined *exactly* without searching, but at considerable expense in the clarity of the code.*

THE ISO CALENDAR

The International Organization for Standardization (ISO) calendar, popular in Sweden and other European countries, specifies a date by giving the ordinal day in the week and the 'calendar week' in a Gregorian year. Section 3.17 of the ISO standard[15] defines a *calendar week* as

> A seven day period within a calendar year, starting on a Monday and identified by its ordinal number within the year; the first calendar week of the year is the one that includes the first Thursday of that year. In the Gregorian calendar, this is equivalent to the week which includes 4 January.

Determining the beginning of the first calendar week of a Gregorian year thus requires determining the Monday on or before January 4 of that year. Since we will need similar determinations for some of the holidays, discussed in a later section, we encapsulate the formula $d - ((d - k) \bmod 7)$ to find the kth day of the week ($k = 0$ for Sunday, and so on) that falls in the seven-day period ending on absolute date d:

```
(defun Kday-on-or-before (date k)
;; Absolute date of the kday on or before date.
;; k = 0 means Sunday, k = 1 means Monday, and so on.
  (- date (mod (- date k) 7)))
```

Applying `Kday-on-or-before` to $d + 6$ gives us the `Kday-on-or-after` an absolute day d. Similarly, applying it to $d + 3$ gives the `Kday-nearest` to absolute date d, applying it to $d - 1$ gives the `Kday-previous` to absolute date d, and applying it to $d + 7$ gives the `Kday-following` absolute date d.

The ISO calendar counts Sunday as the seventh day of the week (throughout this paper we have otherwise counted it as the zeroth day of the week), so we implement this calendar as follows:

---

*The exact determination of the Gregorian year from the absolute *date* is an exercise in base conversion in a mixed-radix system[14]:

$n_{400} = \lfloor (date - 1)/146097 \rfloor$ {number of completed 400 year cycles}
$d_1 = (date - 1) \bmod 146097$ {days not included in n_{400}}
$n_{100} = \lfloor (d_1 - 1)/36524 \rfloor$ {number of 100 year cycles not included in n_{400}}
$d_2 = (d_1 - 1) \bmod 36524$ {days not included in n_{400} or n_{100}}
$n_4 = \lfloor (d_2 - 1)/1461 \rfloor$ {number of 4 year cycles not included in n_{400} or n_{100}}
$d_3 = (d_2 - 1) \bmod 1461$ {days not included in n_{400}, n_{100}, or n_4}
$n_1 = \lfloor (d_3 - 1)/365 \rfloor$ {number of years not included in n_{400}, n_{100}, or n_4}
$d_4 = (d_3 - 1) \bmod 365$ {days not included in n_{400}, n_{100}, n_4, or n_1}

date is ordinal day d_4 in Gregorian year $400 \times n_{400} + 100 \times n_{100} + 4 \times n_4 + n_1 + 1$

Similar calculations can be used for the Julian, Islamic, or Hebrew calendars.

```
(defun absolute-from-iso (date)
;; Absolute date equivalent to ISO date = (week day year).
  (let* ((week (first date))
         (day (second date))
         (year (third date)))
;;  Return
    (+ (Kday-on-or-before
        (absolute-from-gregorian (list 1 4 year))
        1)                          ;;  Days in prior years.
       (* 7 (1- week))              ;;  Days in prior weeks this year.
       (1- day))))                  ;;  Prior days this week.


(defun iso-from-absolute (date)
;; ISO (week day year) corresponding to the absolute date.
  (let* ((approx
          (extract-year (gregorian-from-absolute (- date 3))))
         (year (if (>= date
                       (absolute-from-iso (list 1 1 (1+ approx))))
                   ;; Then
                      (1+ approx)
                   ;; Else
                      approx))
         (week (1+ (quotient
                    (- date (absolute-from-iso (list 1 1 year)))
                    7)))
         (day (if (= 0 (mod date 7))
                  ;; Then
                     7
                  ;; Else
                     (mod date 7))))
 ;;  Return
    (list week day year)))
```

THE JULIAN CALENDAR

The calculations for the Julian calendar, which we described in our discussion of the Gregorian calendar, are nearly identical to those for the Gregorian calendar, but we must change the leap-year rule used in determining the last day of a month:

```
(defun last-day-of-julian-month (month year)
;; Last day in Julian month during year.
  (if ;; February in a leap year
      (and (= month 2) (= (mod year 4) 0))
;; Then return
      29
```

```
    ;; Else return
        (nth (1- month) (list 31 28 31 30 31 30 31 31 30 31 30 31))))
```

Converting from a Julian date to an absolute date requires a calculation similar to that in the Gregorian case, but with two minor adjustments: We no longer need consider century-year leap days, but we subtract 2 because absolute date 1 is January 3, 1 C.E. (Julian), and so the first two days of 1 C.E. (Julian) must be excluded.

```
(defun absolute-from-julian (date)
;; Absolute date equivalent to Julian date.
  (let* ((month (extract-month date))
         (year (extract-year date)))
;;  Return
    (+ (extract-day date)     ;; Days so far this month.
       (sum                   ;; Days in prior months this year.
        (last-day-of-julian-month m year) m 1 (< m month))
       (* 365 (1- year))      ;; Days in prior years.
       (quotient (1- year) 4);; Leap days in prior years.
       -2)))                  ;; Days elapsed before absolute date 1.
```

Except for obvious changes in reference from Gregorian to Julian, conversion of absolute dates to Julian dates is identical to conversion of absolute dates to Gregorian dates.

```
(defun julian-from-absolute (date)
;; Julian (month day year) corresponding to absolute date.
  (let*
      ((approx       ;; Approximation from below.
        (quotient (+ date 2) 366))
       (year         ;; Search forward from the approximation.
        (+ approx
           (sum 1 y approx
                (>= date
                    (absolute-from-julian (list 1 1 (1+ y)))))))
       (month        ;; Search forward from January.
        (1+ (sum 1 m 1
                 (> date
                    (absolute-from-julian
                     (list m
                           (last-day-of-julian-month m year)
                           year))))))
       (day          ;; Calculate the day by subtraction.
        (- date (1- (absolute-from-julian (list month 1 year))))))
;;  Return
    (list month day year)))
```

THE ISLAMIC CALENDAR

The Islamic calendar[4–6, 16, 17] is a straightforward, strictly lunar calendar. Its independence of the solar cycle means that its months do not occur in fixed seasons, but migrate through the solar year. Days begin at sunset.

The calendar is computed, by the majority of the Moslem world, starting at sunset of Thursday, July 15, 622 C.E. (Julian), the year of Mohammed's flight to Medina. In essence, Moslems count absolute date 227015 = Friday, July 16, 622 C.E. (Julian) as the beginning of the Islamic year 1, that is, as Muharram 1, 1 A.H. There are 12 Islamic months which contain, alternately, 29 or 30 days:

(1) Muharram	30 days	(7) Rajab	30 days
(2) Safar	29 days	(8) Sha'ban	29 days
(3) Rabi I	30 days	(9) Ramadan	30 days
(4) Rabi II	29 days	(10) Shawwal	29 days
(5) Jumada I	30 days	(11) Dhu al-Qada	30 days
(6) Jumada II	29 days	(12) Dhu al-Hijjah	29 {30} days

The leap-year structure is given in curly brackets—the last month, Dhu al-Hijjah, contains 30 days in the 2nd, 5th, 7th, 10th, 13th, 16th, 18th, 21st, 24th, 26th, and 29th years of a 30-year cycle.* This gives an average month of 29.5305555··· days. The cycle of common and leap years can be expressed concisely (but obfuscatingly!) by observing that an Islamic year y is a leap year if and only if $(11y + 14) \bmod 30$ is less than 11.

Determining the last day of an Islamic month is thus done by

```
(defun islamic-leap-year (year)
;; True if year is an Islamic leap year.
  (< (mod (+ 14 (* 11 year)) 30) 11))

(defun last-day-of-islamic-month (month year)
;; Last day in month during year on the Islamic calendar.
  (if (or (oddp month)
          (and (= month 12) (islamic-leap-year year)))
;; Then return
      30
;; Else return
    29))
```

The function oddp tests for odd integers. (It would be more efficient to compute `islamic-leap-year` by looking up the value in a 30-bit table.)

Converting from an Islamic date to an absolute date is done by summing the days so far in the current month, the days so far in the current Islamic year, the non-leap days in prior Islamic years, the leap days in prior Islamic years, and the days prior to the Islamic calendar.

*A minority of Moslems have a slightly different leap year structure.

```
(defun absolute-from-islamic (date)
;; Absolute date equivalent to Islamic date.
  (let* ((month (extract-month date))
         (year (extract-year date)))
    (+ (extract-day date)      ;; Days so far this month.
       (* 29 (1- month))       ;; Days so far...
       (quotient month 2)      ;;            ...this year.
       (* (1- year) 354)       ;; Non-leap days in prior years.
       (quotient               ;; Leap days in prior years.
        (+ 3 (* 11 year)) 30)
       227014)))               ;; Days before start of calendar.
```

Computing the Islamic date equivalent to a given absolute date is done almost identically to the computations for the Gregorian and Julian calendars: We approximate the year and search linearly for the exact value; then we find the month by linear search and the day of the month by subtraction.

```
(defun islamic-from-absolute (date)
;; Islamic date (month day year) corresponding to absolute date.
  (if ;; Pre-Islamic date.
      (<= date 227014)
;; Then return
      (list 0 0 0)
;; Else
    (let* ((approx          ;; Approximation from below.
            (quotient (- date 227014) 355))
           (year            ;; Search forward from the approximation.
            (+ approx
               (sum 1 y approx
                    (>= date
                        (absolute-from-islamic
                         (list 1 1 (1+ y)))))))
           (month             ;; Search forward from Muharram.
            (1+ (sum 1 m 1
                     (> date
                        (absolute-from-islamic
                         (list m
                               (last-day-of-islamic-month m year)
                               year))))))
           (day             ;; Calculate the day by subtraction.
            (- date (1- (absolute-from-islamic
                         (list month 1 year))))))
      ;; Return
        (list month day year))))
```

It is important to realize that, to some extent, the above calculations are merely hypothetical because there are many disparate forms of the Islamic calendar.[4] Furthermore, much of the Islamic world relies not on such calculations at all, but on proclamation of the new moon by religious authorities. Consequently, the dates given by the Lisp functions here can be in error by a day or two from what will actually be observed in various parts of the Islamic world; this is unavoidable.

THE HEBREW CALENDAR

The Hebrew calendar[4–6, 16, 18–25] promulgated by Hillel II in the mid-fourth century, is by far the most complicated of the five calendars that we consider. Its complexity is inherent in the requirement that calendar months must be strictly lunar while Passover must always occur in the spring. Since the seasons are dependent on the solar year, the Hebrew calendar must harmonize simultaneously with both lunar and solar events. As in the Islamic calendar, days begin at sunset.

The Hebrew year consists of twelve months in a common year and thirteen in a leap year:

(1) Nisan	30 days	(7) Tishri	30 days
(2) Iyyar	29 days	(8) Ḥeshvan	29 or 30 days
(3) Sivan	30 days	(9) Kislev	29 or 30 days
(4) Tammuz	29 days	(10) Teveth	29 days
(5) Av	30 days	(11) Shevat	30 days
(6) Elul	29 days	{(12) Adar I	30 days}
		(12) {(13)} Adar {II}	29 days

The leap-year structure is given in curly brackets—in a leap year there is an interpolated twelfth month of 30 days called 'Adar I' to distinguish it from the final month, 'Adar II'. The length of the eighth and ninth months vary from year to year according to criteria that will be explained below. Our ordering of the Hebrew months follows biblical custom (Leviticus 23:5) in which (what was later called) Nisan is the first month. This numbering causes the Hebrew new year (Rosh HaShanah) to begin on the first of Tishri which, by our ordering, is the seventh month—but this too agrees with biblical usage (Leviticus 23:24). Adding up the lengths of the months, we see that a normal year has 353–355 days, whereas a leap year has 383–385 days.

The so-called *Metonic cycle* is based on the observation that 19 mean solar years contain almost exactly 235 lunar months. This correspondence, known to ancient Babylonian astronomers, makes a solar/lunar calendar feasible. The $235 = 12 \times 12 + 7 \times 13$ months in the cycle are divided into twelve years of twelve months and seven years of thirteen months. The Metonic cycle is used in the Hebrew calendar and also for the calculation of Easter (as we discuss in the section on holidays).

In the Hebrew calendar, leap years occur in the 3rd, 6th, 8th, 11th, 14th, 17th, and 19th years of the 19-year cycle. As in the Islamic leap-year structure, this sequence can be computed concisely by noting that Hebrew year y is a leap year if and only if $(7y + 1) \bmod 19$ is less than 7. Thus we determine whether a year is a Hebrew leap year by

```
(defun hebrew-leap-year (year)
;; True if year is a leap year.
  (< (mod (1+ (* 7 year)) 19) 7))

(defun last-month-of-hebrew-year (year)
;; Last month of Hebrew year.
  (if (hebrew-leap-year year)
;; Then return
      13
;; Else return
    12))
```

The number of days in a Hebrew month is a more complex issue. The twelfth month, Adar or Adar I, has 29 days in a common year and 30 days in a leap year, but the numbers of days in the eighth month (Ḥeshvan) and ninth month (Kislev) depend on the overall length of the year, which in turn depends on factors discussed below. Thus we write

```
(defun last-day-of-hebrew-month (month year)
;; Last day of month in Hebrew year.
  (if (or (member month (list 2 4 6 10 13))
          (and (= month 12) (not (hebrew-leap-year year)))
          (and (= month 8) (not (long-heshvan year)))
          (and (= month 9) (short-kislev year)))
;; Then return
      29
;; Else return
    30))
```

where the functions `long-heshvan` and `short-kislev` will be given later.

To present the remainder of the calculations for the Hebrew calendar, it is necessary to describe how Hebrew intervals of time are reckoned. The *day* is divided into 24 *hours* and each hour is divided into 1080 *parts*; a day thus has 25920 parts. For our purposes, it is easier to use just days and parts.

The beginning of the Hebrew new year is determined by the occurrence of the new moon (mean conjunction) of the seventh month (Tishri), subject to possible postponements of a day or two. The average length of a lunar period is taken to be 29 days, 12 hours, and 793 parts, that is, approximately 29.530594 days. The new moon of Tishri, 1 A.M. (the first day of the first year for the Hebrew calendar) is fixed at Sunday night, 5 hours, 204 parts. Thus we calculate the time elapsed from sunset of the preceding Saturday evening, until the new moon of Tishri for the Hebrew year y by computing.

$$(1 \text{ day}, 5604 \text{ parts}) + (29 \text{ days}, 13753 \text{ parts}) \times (\text{number of months before year } y).$$

The start of each new year, Rosh HaShanah (Tishri 1), coincides with the calculated day of the mean conjunction of Tishri, unless one of four delays is mandated:

- If the mean conjunction is at midday or after, then the new year is delayed.* Sunset is presumed to occur always at 6 p.m.† Since there are then 18 hours from sunset until midday, postponement occurs if the conjunction is at $18 \times 1080 = 19440$ parts or later into the day.

- In no event must the new year be on Sunday, Wednesday, or Friday.‡ This introduces an average 'correction' of about half a day in the calculated time of appearance of the new moon of the month of Tishri.[22]

- To keep the length of a year within the allowable ranges, in rare cases, an additional delaying factor may need to be employed. If Rosh HaShanah were on Tuesday and the conjunction of the following year were at midday or later, then applying the previous two rules would result in delaying the following Rosh HaShanah from Saturday (the day of the next conjunction for a common year) until Monday. This would require an (unacceptable) year length of 356 days, so instead the current Rosh HaShanah is delayed until Thursday.

- Rosh HaShanah on Monday after a leap year can pose a similar problem, by causing the year just ending to be too short. In this case, Rosh HaShanah is delayed until Tuesday.

These rules are perhaps best described algorithmically:

```
(defun hebrew-calendar-elapsed-days (year)
;; Number of days elapsed from the Sunday prior to the start of the
;; Hebrew calendar to the mean conjunction of Tishri of Hebrew year.
  (let*
      ((months-elapsed
        (+
         (* 235              ;; Months in complete cycles so far.
            (quotient (1- year) 19))
         (* 12               ;; Regular months in this cycle.
            (mod (1- year) 19))
         (quotient           ;; Leap months this cycle
          (1+ (* 7 (mod (1- year) 19)))
          19)))
       (parts-elapsed (+ 5604 (* 13753 months-elapsed)))
       (day                                    ;; Conjunction day
        (+ 1 (* 29 months-elapsed) (quotient parts-elapsed 25920)))
       (parts (mod parts-elapsed 25920))       ;; Conjunction parts
```

*In 923 C.E. the calculated conjunction fell just after midday; this caused a short-lived (921–923 C.E.) dispute between Palestinian and Babylonian Jewish authorities about whether this rule should be applied; some scant details can be found in vol. 4, col. 539–540 of the *Encyclopædia Judaica*.[26]

†That is, the moment of sunset is *deemed* 6 p.m. and sunrise is *deemed* 6 a.m., so that the 'daylight hours' and 'nighttime hours' have different lengths that vary according to the seasons. (This is the correct interpretation of chap. 6, par. 2 in Maimonides' code.[22])

‡Excluding Wednesday and Friday prevents Yom Kippur (Tishri 10) from falling on Friday or Sunday; excluding Sunday prevents Hoshanah Rabba (Tishri 21) from falling on Saturday.

```
      (alternative-day
       (if (or
            (>= parts 19440)  ;; If new moon is at or after midday,
            (and
              (= (mod day 7) 2);; ...or is on a Tuesday...
              (>= parts 9924)  ;; at 9 hours, 204 parts or later...
              (not (hebrew-leap-year year)));; of a common year,
            (and
              (= (mod day 7) 1);; ...or is on a Monday at...
              (>= parts 16789) ;; 15 hours, 589 parts or later...
              (hebrew-leap-year;; at the end of a leap year
               (1- year))))
         ;; Then postpone Rosh HaShanah one day
           (1+ day)
         ;; Else
           day)))
  (if ;; If Rosh HaShanah would occur on Sunday, Wednesday,
      ;; or Friday
      (member (mod alternative-day 7) (list 0 3 5))
;; Then postpone it one (more) day and return
      (1+ alternative-day)
;; Else return
     alternative-day)))
```

Although the calculations as given above are correct, they involve intermediate values that can be far larger than 2^{24} for current dates. To avoid this problem we can modify the computation of the day and parts of the conjunction in the `let*` to read

```
      ⋮
      (parts-elapsed
       (+ 204
          (* 793 (mod months-elapsed 1080))))
      (hours-elapsed
       (+ 5
          (* 12 months-elapsed)
          (* 793 (quotient months-elapsed 1080))
          (quotient parts-elapsed 1080)))
      (day                                  ;; Conjunction day
       (+ 1
          (* 29 months-elapsed)
          (quotient hours-elapsed 24)))
      (parts                                ;; Conjunction parts
       (+ (* 1080 (mod hours-elapsed 24))
          (mod parts-elapsed 1080)))
      ⋮
```

With this modification, only 24 bits are necessary for dates in the foreseeable future.

As mentioned above, the length of the year determines the length of the two varying months, Ḥeshvan and Kislev. Ḥeshvan is long (30 days) if the year has 355 or 385 days; Kislev is short (29 days) if the year has 353 or 383 days. The length of the year, in turn, is determined by the dates of the Hebrew new years (Tishri 1) preceding and following the year in question:

```
(defun days-in-hebrew-year (year)
;; Number of days in Hebrew year.
  (- (hebrew-calendar-elapsed-days (1+ year))
     (hebrew-calendar-elapsed-days year)))

(defun long-heshvan (year)
;; True if Heshvan is long in Hebrew year.
  (= (mod (days-in-hebrew-year year) 10) 5))

(defun short-kislev (year)
;; True if Kislev is short in Hebrew year.
  (= (mod (days-in-hebrew-year year) 10) 3))
```

With all the above machinery, we are now ready to convert to and from Hebrew dates, in a manner similar to the previous calendars:

```
(defun absolute-from-hebrew (date)
;; Absolute date of Hebrew date.
  (let* ((month (extract-month date))
         (day (extract-day date))
         (year (extract-year date)))
;; Return
    (+ day                                  ;; Days so far this month.
       (if ;; before Tishri
           (< month 7)
        ;; Then add days in prior months this year before and
        ;; after Nisan.
           (+ (sum (last-day-of-hebrew-month m year)
                   m 7 (<= m (last-month-of-hebrew-year year)))
              (sum (last-day-of-hebrew-month m year)
                   m 1 (< m month)))
        ;; Else add days in prior months this year
           (sum (last-day-of-hebrew-month m year) m 7 (< m month)))
       (hebrew-calendar-elapsed-days year);; Days in prior years.
       -1373429)))                  ;; Days elapsed before absolute date 1.


(defun hebrew-from-absolute (date)
;; Hebrew (month day year) corresponding to absolute date.
  (let* ((approx    ;; Approximation from below.
          (quotient (+ date 1373429) 366))
         (year      ;; Search forward from the approximation.
```

```
        (+ approx (sum i y approx
                       (>= date
                           (absolute-from-hebrew
                            (list 7 1 (1+ y)))))))
      (start      ;; Starting month for search for month.
       (if (< date (absolute-from-hebrew (list 1 1 year)))
           ;; Then start at Tishri
               7
           ;; Else start at Nisan
              1))
      (month      ;; Search forward from either Tishri or Nisan.
       (+ start
          (sum i m start
               (> date
                  (absolute-from-hebrew
                   (list m
                         (last-day-of-hebrew-month m year)
                         year))))))
      (day        ;; Calculate the day by subtraction.
       (- date (1- (absolute-from-hebrew (list month 1 year))))))
;; Return
    (list month day year)))
```

The function `hebrew-calendar-elapsed-days` is called repeatedly during the calculations, often several times for the same year. More efficient code would avoid such repetition.

HOLIDAYS

The various calendars are needed to compute the dates of civil and religious holidays. In most of this section, we will take the ethnocentric view that our task is to compute the absolute dates of holidays that occur in a given Gregorian year; there is clearly little difficulty in finding the dates of, say, Islamic holidays in a given Islamic year!

Secular Holidays

Secular holidays on the Gregorian calendar are either on fixed days or on a particular day of the week relative to the beginning or end of a month. (An extensive list of secular holidays can be found in Gregory's *Special Days.*[27]) Fixed holidays are trivial to deal with; for example, to determine the absolute date of American Independence Day in a given Gregorian year we would use

```
(defun independence-day (year)
;; Absolute date of American Independence Day in Gregorian year.
  (absolute-from-gregorian (list 7 4 year)))
```

Other holidays are on the nth occurrence of a given day of the week, counting from either the beginning or the end of the month. American Labor Day, for example, is the

first Monday in September, while American Memorial Day is the last Monday in May. We handle such specifications by writing a function that determines the absolute date of the nth kth day in a given month in a given Gregorian year, counting backward from the end of the month when $n < 0$.

```
(defun Nth-Kday (n k month year)
;; Absolute date of the nth kday in Gregorian month, year.
;; If n<0, the nth kday from the end of month is returned
;; (that is, -1 is the last kday, -2 is the penultimate kday,
;; and so on).  k = 0 means Sunday, k = 1 means Monday, and so on.
  (if (> n 0)
;; Then return
      (+ (Kday-on-or-before                    ;; First kday in month.
          (absolute-from-gregorian
           (list month 7 year)) k)
         (* 7 (1- n)))                         ;; Advance n - 1 kdays.
;; Else return
      (+ (Kday-on-or-before                    ;; Last kday in month.
          (absolute-from-gregorian
           (list month
                 (last-day-of-gregorian-month month year)
                 year))
          k)
         (* 7 (1+ n)))))                       ;; Go back -n - 1 kdays.
```

With this function, we can define holiday dates, such as

```
(defun labor-day (year)
;; Absolute date of American Labor Day in Gregorian year.
  (Nth-Kday 1 1 9 year));; First Monday in September.

(defun memorial-day (year)
;; Absolute date of American Memorial Day in Gregorian year.
  (Nth-Kday -1 1 5 year));; Last Monday in May.
```

or determine the starting and ending dates of American daylight savings time:

```
(defun daylight-savings-start (year)
;; Absolute date of the start of American daylight savings time
;; in Gregorian year.
  (Nth-Kday 1 0 4 year));; First Sunday in April.

(defun daylight-savings-end (year)
;; Absolute date of the end of American daylight savings time
;; in Gregorian year.
  (Nth-Kday -1 0 10 year));; Last Sunday in October.
```

Christian Holidays

The main Christian holidays are Christmas, Easter, and various days connected with them (Advent, Ash Wednesday, Good Friday, and others; see vol. V, pp. 844–853 of the *Encyclopædia of Religion and Ethics.*[7]) In addition to the complicated calculations necessary to determine the date of Easter, there are complications that result from the Eastern Orthodox practice of celebrating Christmas according to the Julian calendar.

The date of Christmas on the Gregorian calendar is fixed and presents no problem:

```
(defun christmas (year)
;; Absolute date of Christmas in Gregorian year.
  (absolute-from-gregorian (list 12 25 year)))
```

The related dates of Advent (Sunday closest to November 30; this is equivalent to the Sunday on or before December 3) and Epiphany (twelve days after Christmas) are computed by

```
(defun advent (year)
;; Absolute date of Advent in Gregorian year.
  (Kday-on-or-before (absolute-from-gregorian (list 12 3 year)) 0))

(defun epiphany (year)
;; Absolute date of Epiphany in Gregorian year.
  (+ 12 (christmas year)))
```

The date of Assumption (August 15), celebrated in Catholic and Eastern Orthodox countries, is fixed, and presents no problem.

The date of Eastern Orthodox Christmas occurring in a given Gregorian year is more involved. Since the Julian year is always at least as long as the corresponding Gregorian year, Eastern Orthodox Christmas can occur at most once in a given Gregorian year, but it can occur either at the beginning or the end; in some years (like 1100 C.E.) it does not occur at all.

```
(defun eastern-orthodox-christmas (year)
;; List of zero or one absolute dates of Eastern Orthodox
;; Christmas in Gregorian year.
  (let* ((jan1 (absolute-from-gregorian (list 1 1 year)))
         (dec31 (absolute-from-gregorian (list 12 31 year)))
         (y (extract-year (julian-from-absolute jan1)))
         (c1 (absolute-from-julian (list 12 25 y)))
         (c2 (absolute-from-julian (list 12 25 (1+ y)))))
    (append
     (if ;; c1 occurs in current year
         (<= jan1 c1 dec31)
;;    Then that date; otherwise, none
         (list c1) nil)
     (if ;; c2 occurs in current year
```

```
        (<= jan1 c2 dec31)
;;   Then that date; otherwise, none
        (list c2) nil))))
```

The function append concatenates its arguments into one list.

The calculation of the date of Easter has a fascinating history.[11, 12] Algorithms and computer programs abound[5, 6, 28–32] (the discussion of the mathematical aspects of the calculation in O'Beirne's *Puzzles and Paradoxes*[31] is especially nice), but we include Lisp functions here for completeness and because our absolute-date approach allows considerable simplification of 'classical' algorithms.

The date of Easter was fixed in 325 C.E. by the Council of Nicæa to be the first Sunday after the first full moon occurring on or after the vernal equinox. This definition seems precise, but in reality accurate determination of the full moon and the vernal equinox is quite complex and simpler approximations are used in practice; as Kepler declared,[12] 'Easter is a feast, not a planet'. The date of Easter is thus based on the presumption that the vernal equinox is always March 21 and on approximations to the lunar phases called *epacts*.

Before the Gregorian reform of the Julian calendar the approximations were fairly crude. Assuming the Metonic cycle were accurate would mean that the phase of the moon on January 1 would be the same every 19 years. Hence, the epact can be approximated by multiplying the number of years since the start of the current Metonic cycle (called the 'golden number') by the eleven-day difference between a common year of 365 days and 12 lunar months of 29.5 days and adjusting by the epact of January 1, 1 C.E.—all this done modulo 30. From the epact one can calculate the phase of the moon (in days) on April 5 (ignoring February 29) and go back that many days from April 19, giving a date between March 21 and April 19 (inclusive) for the (ecclesiastical) 'Paschal full moon'. Thus the equivalent of the following calculation was used to determine Easter from 325 C.E. until the adoption of the Gregorian calendar:[5, 6, 29]

```
(defun nicaean-rule-easter (year)
;; Absolute date of Easter in Julian year, according to the rule
;; of the Council of Nicaea.
  (let* ((shifted-epact ;; Age of moon for April 5.
          (mod (+ 14
                  (* 11 (mod year 19)))
               30))
         (paschal-moon  ;; Day after full moon on or after March 21.
          (- (absolute-from-julian (list 4 19 year))
             shifted-epact)))
;; Return the Sunday following the Paschal moon
    (Kday-on-or-before (+ paschal-moon 7) 0)))
```

The Gregorian reform included, for the calculation of Easter, a far more accurate approximation to the lunar phases due to Clavius. Three corrections are employed:

- The Gregorian change in the calendar (three out of four century years are common years) is taken into account in the calculation of epacts.
- About every 222 years, a one-day correction is made to compensate for the inaccuracy of the Metonic cycle.
- There are approximately 29.5 days between successive new moons, while the lunar month in which Easter occurs is always taken to have 29 days, so an additional adjustment to the epact (which can take on 30 values) is sometimes needed.*

This is the method now used:[5, 6, 30]

```
(defun easter (year)
;; Absolute date of Easter in Gregorian year.
  (let* ((century (1+ (quotient year 100)))
         (shifted-epact        ;; Age of moon for April 5...
          (mod
           (+ 14 (* 11 (mod year 19));;      ...by Nicaean rule
              (-       ;; ...corrected for the Gregorian century rule
               (quotient (* 3 century) 4))
              (quotient;; ...corrected for Metonic cycle inaccuracy.
               (+ 5 (* 8 century)) 25)
              (* 30 century));;              Keeps value positive.
           30))
         (adjusted-epact       ;;  Adjust for 29.5 day month.
          (if (or (= shifted-epact 0)
                  (and (= shifted-epact 1) (< 10 (mod year 19))))
        ;; Then
              (1+ shifted-epact)
        ;; Else
            shifted-epact))
         (paschal-moon;; Day after full moon on or after March 21.
          (- (absolute-from-gregorian (list 4 19 year))
             adjusted-epact)))
;; Return the Sunday following the Paschal moon.
    (Kday-on-or-before (+ paschal-moon 7) 0)))
```

Many Christian holidays depend on the date of Easter: Septuagesima Sunday (63 days before), Sexagesima Sunday (56 days before), Shrove Sunday (49 days before), Shrove Monday (48 days before), Shrove Tuesday (47 days before), Ash Wednesday (46 days before), Passion Sunday (14 days before), Palm Sunday (seven days before), Maundy Thursday (three days before), Good Friday (two days before), Rogation Sunday (35 days after), Ascension Day (39 days after), Pentecost (also called

*The details of this adjustment were designed by Clavius to keep Easter within the same range of dates as in the Julian calendar, March 22 through April 25.[31]

Whitsunday—49 days after), Whitmundy (50 days after), Trinity Sunday (56 days after), and Corpus Christi (60 days after). All these are easily computed; for example

```
(defun pentecost (year)
;; Absolute date of Pentecost in Gregorian year.
  (+ 49 (easter year)))
```

Islamic Holidays

Determining the absolute dates of Islamic holidays occurring in a given Gregorian year is complicated because an Islamic year is always shorter than the Gregorian year, so each Gregorian year contains parts of at least two and sometimes three successive Islamic years. Hence any given Islamic date occurs at least once and possibly twice in any given Gregorian year. For example, Islamic New Year (Muharram 1) occurred twice in 1943: on January 8 and again on December 28. Accordingly, we will approach the problem of the Islamic holidays by writing a general function to return a list of the absolute dates of a given Islamic date occurring in a given Gregorian year:

```
(defun islamic-date (month day year)
;; List of the absolute dates of Islamic month, day
;; that occur in Gregorian year.
  (let* ((jan1 (absolute-from-gregorian (list 1 1 year)))
         (dec31 (absolute-from-gregorian (list 12 31 year)))
         (y (extract-year (islamic-from-absolute jan1)))
;; The possible occurrences in one year are
         (date1 (absolute-from-islamic (list month day y)))
         (date2 (absolute-from-islamic (list month day (1+ y))))
         (date3 (absolute-from-islamic (list month day (+ 2 y)))))
;; Combine in one list those that occur in current year
    (append
      (if (<= jan1 date1 dec31)
          (list date1) nil)
      (if (<= jan1 date2 dec31)
          (list date2) nil)
      (if (<= jan1 date3 dec31)
          (list date3) nil))))
```

There is little uniformity among the Islamic sects and countries as to holidays. In general, the principal holidays of the Islamic year are Islamic New Year (Muharram 1), Ashura (Muharram 10), Mulad-al-Nabi (Rabi I 12), Shab-e-Mi'raj (Rajab 26), Shab-e-Bara't (Sha'ban 15), Ramadan (Ramadan 1), Shab-e Qadr (Ramadan 27), Id-al-Fitr (Shawwal 1), and Id-al-Adha (Dhu-al-Hijjah 10).* Like all Islamic days, an Islamic holiday begins at sunset the prior evening. We can determine a list of

*Other days, too, have religious significance, for example, the entire month of Ramadan.

the corresponding absolute dates of occurrence in a given Gregorian year by using `islamic-date` above, as in

```
(defun mulad-al-nabi (year)
;; List of absolute dates of Mulad-al-Nabi occurring in
;; Gregorian year.
  (islamic-date 3 12 year))
```

Islamic holidays begin at sunset on the prior evening. It bears reiterating that the determination of the Islamic holidays cannot be fully accurate since the precise day of their occurrence is dependent on proclamation by religious authorities.

Jewish Holidays

As throughout this section, we consider our problem to be the determination of the Jewish holidays that occur in a specified Gregorian year. Since the Hebrew year is, on the average, consistently aligned with the Gregorian year, each Jewish holiday occurs just once in a given Gregorian year (with a minor exception noted below). The major holidays of the Hebrew year occur on fixed days on the Hebrew calendar, but occur only in fixed seasons on the Gregorian calendar.[7, 24] They are easy to determine on the Gregorian calendar with the machinery developed above, provided we observe that the Hebrew year that began in the fall of 1 C.E. (Gregorian) was 3762 A.M., so we have the relation

$$y + 3761 = \text{Hebrew new year occurring in the fall of Gregorian year } y.$$

This means that holidays occurring in the fall and early winter of the Gregorian year y occur in the Hebrew year $y + 3761$, while holidays in the late winter, spring, and summer occur in Hebrew year $y + 3760$. For example, to find the absolute date of Yom Kippur (Tishri 10) in a Gregorian year, we would use

```
(defun yom-kippur (year)
;; Absolute date of Yom Kippur occurring in Gregorian year.
  (absolute-from-hebrew (list 7 10 (+ year 3761))))
```

The absolute dates of Rosh HaShanah (Tishri 1), Sukkot (Tishri 15), Hoshanah Rabba (Tishri 21), Shemini Azereth (Tishri 22), and Simḥat Torah (Tishri 23, outside Israel) are identically determined. (See p. 800 of *Winning Ways, Volume 2: Games in Particular*[28] for another way to determine the date of Rosh HaShanah.) As on the Islamic calendar, all Hebrew holidays begin at sunset the prior evening.

The dates of the other major holidays, Passover (Nisan 15), ending of Passover (Nisan 21), and Shavuot (Sivan 6), are determined similarly, but since these holidays occur in the spring, the year corresponding to Gregorian year y is $y + 3760$.* Thus, for example, we determine the absolute date of Passover by

*Conservative and Orthodox Jews observe two days Rosh HaShanah—Tishri 1 and 2. Outside Israel, they also observe Tishri 16, Nisan 16, Nisan 22, and Sivan 7 as holidays.

```
(defun passover (year)
;; Absolute date of Passover occurring in Gregorian year.
  (absolute-from-hebrew (list 1 15 (+ year 3760))))
```

(Gauss developed an interesting formula to determine the Gregorian date of Passover in a given year.[20])

The minor holidays of the Hebrew year are the 'intermediate' days of Sukkot (Tishri 16–21) and Passover (Nisan 16–20), Ḥanukkah (eight days, beginning on Kislev 25), Tu-B'Shevat (Shevat 15), and Purim (Adar 14 in normal years, Adar II 14 in leap years). Ḥanukkah occurs in late fall/early winter, so Ḥanukkah of Gregorian year y occurs in the Hebrew year $y + 3761$, while Tu-B'Shevat occurs in late winter/early spring and hence Tu-B'Shevat of Gregorian year y occurs in Hebrew year $y+3760$; thus these two holidays are handled as were Rosh HaShanah and Passover, respectively. Purim always occurs in late winter or early spring, so its absolute date is computed by

```
(defun purim (year)
;; Absolute date of Purim occurring in Gregorian year.
  (absolute-from-hebrew
    (list
      (last-month-of-hebrew-year (+ year 3760));; Adar or Adar II
      14
      (+ year 3760))))
```

The Hebrew year contains several fast days that, though specified by particular Hebrew calendar dates, are shifted when those days occur on Saturday. The fast days are Tzom Gedaliah (Tishri 3), Tzom Teveth (Teveth 10), Ta'anit Esther (the day before Purim), Tzom Tammuz (Tammuz 17), and Tisha B'Av (Av 9). When Purim is on Sunday, Ta'anith Esther occurs on the preceding Thursday, so we can write

```
(defun ta-anit-esther (year)
;; Absolute date of Purim occurring in Gregorian year.
  (let* ((purim-date (purim year)))
    (if ;; Purim is on Sunday
        (= (mod purim-date 7) 0)
  ;; Then return prior Thursday
        (- purim-date 3)
  ;; Else return previous day
      (1- purim-date))))
```

Each of the other fast days, as well as Shushan Purim (the day after Purim, celebrated in Jerusalem), is postponed to the following day (Sunday) when it occurs on Saturday. Since Tzom Gedaliah is always in the fall, while Tzom Tammuz and Tisha B'Av are always in the summer, their determination is easy. For example,

```
(defun tisha-b-av (year)
;; Absolute date of Tisha B'Av occurring in Gregorian year.
  (let* ((ninth-of-av
          (absolute-from-hebrew (list 5 9 (+ year 3760)))))
    (if ;; Ninth of Av is Saturday
        (= (mod ninth-of-av 7) 6)
    ;; Then return the next day
        (1+ ninth-of-av)
    ;; Else return
       ninth-of-av)))
```

Tzom Teveth, which can never occur on Saturday, should be handled like Islamic holidays because Teveth 10 can fall on either side of January 1, so a single Gregorian calendar year can have zero, one, or two occurrences of Tzom Teveth. For example, Tzom Teveth occurred twice in 1982, but not at all in 1984. We leave it to the reader to work out the details.

On the Hebrew calendar, the first day of each month except Tishri is called Rosh Ḥodesh and has minor ritual significance. When the preceding month has 30 days, Rosh Ḥodesh includes also the last day of the preceding month. The determination of these days is elementary. Some other dates of significance depend on the Julian approximation of the tropical year in which each of the four seasons is taken to be 91.3125 days long.*

Finally, the Hebrew calendar contains what we might term 'personal' days: One's birthday according to the Hebrew calendar determines the day of one's *Bat* (for girls) or *Bar* (for boys) *Mitzvah* (the twelfth or thirteenth birthday). Dates of death determine when *Kaddish* is recited (*yahrzeits*) for parents (and sometimes for other relatives). These are ordinarily just anniversary dates, but the leap year structure and the varying number of days in some months require that alternative days be used in certain years, just as someone born February 29 on the Gregorian calendar has to substitute an alternate day in common years.

The particular alternatives are best described algorithmically; since the problem here is the determination of a date in a given Hebrew year, we write the functions in that way:†

```
(defun hebrew-birthday (birthdate year)
;; Absolute date of the anniversary of Hebrew birthdate
;; occurring in Hebrew year.
  (let* ((birth-day (extract-day birthdate))
         (birth-month (extract-month birthdate))
```

*By one traditional Hebrew reckoning, the vernal equinox of year 5685 A.M. was at 6 p.m. Wednesday evening, March 26, 1925 C.E. (Julian). It recurs on that day of the Julian calendar and at that hour of the week every 28 years (*birkhat haḥama*). The beginning of *sh'ela* (request for rain) outside Israel in Julian year y (meant to correspond to 60 days after the autumnal equinox) is absolute day (- (absolute-from-julian (list 3 26 (1+ y))) 124). See Spier's *The Comprehensive Hebrew Calendar*.[24]

†There are minor variations in custom regarding the date of *yahrzeit* in some of these cases, but they need not concern us here.

```
        (birth-year (extract-year birthdate)))
    (if ;; It's Adar in a normal year or Adar II in a leap year,
        (= birth-month (last-month-of-hebrew-year birth-year))
  ;; Then use the same day in last month of year.
        (absolute-from-hebrew
         (list (last-month-of-hebrew-year year) birth-day year))
  ;; Else use the normal anniversary of the birth date,
  ;; or the corresponding day in years without that date
      (absolute-from-hebrew (list birth-month birth-day year)))))
```

This code takes advantage of the fact that absolute-from-hebrew works for dates (*month i year*) and always returns the absolute date of the $(i-1)$st day after (*month* 1 *year*), even if the month has fewer than i days.

```
(defun yahrzeit (death-date year)
;; Absolute date of the anniversary of Hebrew death-date
;; occurring in Hebrew year.
  (let* ((death-day (extract-day death-date))
         (death-month (extract-month death-date))
         (death-year (extract-year death-date)))
    (cond
     ;; If it's Heshvan 30 it depends on the first anniversary; if
     ;; that was not Heshvan 30, use the day before Kislev 1.
     ((and (= death-month 8)
           (= death-day 30)
           (not (long-heshvan (1+ death-year))))
      (1- (absolute-from-hebrew (list 9 1 year))))
     ;; If it's Kislev 30 it depends on the first anniversary; if
     ;; that was not Kislev 30, use the day before Teveth 1.
     ((and (= death-month 9)
           (= death-day 30)
           (short-kislev (1+ death-year)))
      (1- (absolute-from-hebrew (list 10 1 year))))
     ;; If it's Adar II, use the same day in last month of
     ;; year (Adar or Adar II).
     ((= death-month 13)
      (absolute-from-hebrew
       (list (last-month-of-hebrew-year year) death-day year)))
     ;; If it's the 30th in Adar I and year is not a leap year
     ;; (so Adar has only 29 days), use the last day in Shevat.
     ((and (= death-day 30)
           (= death-month 12)
           (not (hebrew-leap-year death-year)))
      (absolute-from-hebrew (list 11 30 year)))
     ;; In all other cases, use the normal anniversary of the
     ;; date of death.
     (t (absolute-from-hebrew
         (list death-month death-day year))))))
```

The cond statement lists a sequence of tests and values, like a generalized case statement.

OTHER CALENDARS

In this section, we briefly describe two calendars for which we would have liked to provide algorithms.

Hindu Calendars

The Hindu calendar is a luni-solar calendar based on approximations to the true astronomical day, synodic month, and sidereal year, rather than to their mean values, as is the case for the calendars considered in previous sections. There are three main variations in use in India; the one based on the *Sūrya-Siddhānta* (circa 1000 C.E.) uses a value of 365.258756481481··· days for the length of the sidereal year. Days of a month are numbered according to the computed phase of the moon at sunrise (in a particular location). Months are named according to the position of the sun in the zodiac at the computed time of new (or, in some variations, full) moon (itself a function of the sun's position). Since the true solar and lunar motions across the celestial sphere vary in speed, this method of reckoning leads occasionally to consecutive days bearing the *same* ordinal number, to skipped numbers, and, similarly, to repeated and lost months.[33, 34]

The necessary computational mechanisms for handling the Hindu calendars are far too complex for inclusion in this paper.[35] Indeed, as van Wijk wrote,[34] 'The rules the Sūrya-Siddhānta gives for calculating the time of true sunrise are exceedingly complicated, and inapplicable in practice.'

The Chinese Calendar

The Chinese year consists of twelve lunar months in a common year and thirteen lunar months in a leap year. Chinese New Year occurs at the beginning of the lunar month during which the sun's tropical longitude is 330 degrees.[36]

The Chinese calendar is similar to the Hindu in its use of true values for solar and lunar events. Unlike the Hindu calendars, the Chinese calendar makers use modern astronomy to decide close calls (whether the sun and moon at the point of true conjunction are in one constellation or in another). The formulae of the ephemeris[9] can be used for this purpose, but require floating point arithmetic, and are not valid over extended periods of time. For these reasons, we have not included Lisp functions for the Chinese calendar or the determination of Chinese New Year in this paper.

ACKNOWLEDGEMENTS

We thank Yaacov Choueka, Douglas Comer, Supyan Hussin, Mitch Medow, Lester Penner, and Daniel Schindler for their comments.

REFERENCES

Note: Only English language references are included in the following list. Some of these are relatively inaccessible.

1. R. M. Stallman, *GNU Emacs Manual*, 6th ed., Free Software Foundation, Cambridge, MA, 1986.
2. G. L. Steele, Jr., COMMON LISP: *The Language*, Digital Press, Boston, MA, 1984.
3. F. Parise, *The Book of Calendars*, Facts on File, New York, 1982.
4. V. V. Tsybulsky, *Calendars of Middle East Countries*, Institute of Oriental Studies, USSR Academy of Sciences, Moscow, 1979.
5. W. S. B. Woolhouse, *Measures, Weights, & Moneys of All Nations: And an Analysis of the Christian, Hebrew, and Mahometan Calendars*, 6th ed., Crosby Lockwood, London, 1881.
6. W. S. B. Woolhouse, 'Calendar', *The Encyclopædia Britannica*, 11th ed., vol. 4, pp. 987–1004, The Encyclopædia Britannica Co., New York, 1910. The same article also appears in the eighth through tenth editions.
7. J. Hastings, ed., *Encyclopædia of Religion and Ethics*, Charles Scribner's Sons, New York, 1911.
8. L. Lamport, 'On the proof of correctness of a calendar program', *Comm. ACM*, **22**, 554–556 (1979).
9. *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*, Her Majesty's Stationery Office, London, 1961.
10. G. V. Coyne, M. A. Hoskin and O. Pedersen, *Gregorian Reform of the Calendar: Proceedings of the Vatican Conference to Commemorate Its 400th Anniversary, 1582–1982*, Pontifica Academica Scientiarum, Specola Vaticana, Vatican, 1983.
11. J. Dutka, 'On the Gregorian revision of the Julian calendar', *Math. Intelligencer*, **10**, 56–64 (1988).
12. G. Moyer, 'The Gregorian calendar', *Sci. Amer.*, **246**, (5), 144–152 (May 1982).
13. V. F. Rickey, 'Mathematics of the Gregorian calendar', *Math. Intelligencer*, **7**, 53–56 (1985).
14. E. M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
15. *Data Elements and Interchange Formats—Information Interchange—Representation of Dates*, ISO 8601, International Organization for Standardization, 1988. This standard replaced ISO 2015, the original document describing the ISO calendar.
16. S. B. Burnaby, *Elements of the Jewish and Muhammadan Calendars*, George Bell and Sons, London, 1901.
17. G. S. P. Freeman-Grenville, *The Muslim and Christian Calendars*, 2nd ed., Rex Collings, London, 1977.
18. N. Bushwick, *Understanding the Jewish Calendar*, Moznaim Publishing Corp., New York, 1989.
19. W. M. Feldman, *Rabbinical Mathematics and Astronomy*, M. L. Cailingold, London, 1931; 3rd corrected ed., Sepher-Hermon, New York, 1978.
20. M. Friedländer, 'Calendar', *The Jewish Encyclopedia*, Funk and Wagnalls, New York, 1906, pp. 501–508.
21. L. Levi, *Jewish Chronomony: The Calendar and Times of Day in Jewish Law*, Gur Aryeh Institute for Advanced Jewish Scholarship, Brooklyn, NY, 1967.
22. Maimonides (= Moshe ben Maimon), *Mishneh Torah: Sefer Zemanim—Hilḥot Kiddush HaḤodesh*, 1177. Translated by S. Gandz (with commentary by J. Obermann and O. Neugebauer), as *Code of Maimonides, Book Three, Treatise Eight, Sanctification of the New Moon*, Yale Judaica Series, vol. XI, Yale University Press, New Haven, CT, 1956. Addenda and corrigenda by E. J. Wiesenberg appear at the end of *Code of Maimonides, Book Three, The Book of Seasons*, translated by S. Gandz and H. Klein, Yale Judaica Series, vol. XIV, Yale University Press, New Haven, CT, 1961.
23. L. A. Resnikoff, 'Jewish calendar calculations', *Scripta Math.*, **9**, 191–195, 274–277 (1943).

24. A. Spier, *The Comprehensive Hebrew Calendar*, 3rd ed., Feldheim Publishers, New York, 1986.
25. E. J. Wiesenberg, 'Calendar', *Encyclopædia Judaica*, vol. 5, col. 43–50, Macmillan, New York, 1971.
26. C. Roth, ed., *Encyclopædia Judaica*, Macmillan, New York, 1971.
27. R. W. Gregory, *Special Days*, Citadel, Secaucus, NJ, 1975. Previous editions appeared under the title, *Anniversaries and Holidays*.
28. E. R. Berlekamp, J. H. Conway and R. K. Guy, *Winning Ways: Volume 2, Games in Particular*, Academic Press, New York, 1982.
29. D. E. Knuth, 'The calculation of Easter', *Comm. ACM*, 5, 209–210 (1962).
30. D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.
31. T. H. O'Beirne, *Puzzles and Paradoxes*, Oxford University Press, London, 1965. Reprinted by Dover Publications, New York, 1984.
32. J. V. Uspensky and M. A. Heaslet, *Elementary Number Theory*, McGraw-Hill, New York, 1939.
33. V. B. Ketkar, 'Indian and foreign chronology: with theory, practice and tables, B.C. 3102 to 2100 A.D. and notices of the Vedic, the ancient Indian, the Chinese, the Jewish, the ecclesiastical and the Coptic calendars', *J. Royal Asiatic Soc., Bombay Branch*, XXVI, (LXXV-A, *Extra Number*), 1923.
34. W. E. van Wijk, *Decimal Tables for the Reduction of Hindu Dates from the Data of the Sūrya-Siddhānta*, Martinus Nijhoff, The Hague, 1938.
35. W. E. van Wijk, 'On Hindu chronology V: Decimal tables for calculating true local time, according to the Sūrya-Siddhānta', *Acta Orientalia*, V, 1–27 (1926).
36. H. Fritsche, *On Chronology and the Construction of the Calendar with Special Regard to the Chinese Computation of Time Compared with the European*, R. Laverentz, St. Petersburg, 1886.

SOFTWARE–PRACTICE AND EXPERIENCE, VOL. 20(9), 929–964 (SEPTEMBER 1990)

The μSystem: Providing Light-weight Concurrency on Shared-memory Multiprocessor Computers Running UNIX

PETER A. BUHR AND RICHARD A. STROOBOSSCHER
Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1

SUMMARY

This paper presents a description of the μSystem, which is a library of C routines that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX operating system. A discussion of the run-time structure of a μSystem program is given, which includes the following concepts: coroutines, tasks, virtual processors and clusters. Next the routines that implement these concepts are discussed in detail. Finally, some performance figures from the μSystem are given and discussed, followed by a comparison of the μSystem with other similar systems.

KEY WORDS Concurrency C Multiprocessor Light-weight process

INTRODUCTION

This paper presents a description of the μSystem,[1] which is a library of C[2] routines that provide light-weight concurrency on uniprocessor and multiprocessor computers running the UNIX* operating system. Concurrent operations in the μSystem are explicitly specified and not inferred from existing constructs in C. Users first design algorithms that are inherently concurrent and then explicitly code corresponding concurrent operations using the routines in the μSystem.

The μSystem uses a shared-memory model of concurrency. This shared-memory is populated by subroutines, coroutines and concurrently executing light-weight processes, called tasks. Coroutine mechanisms are provided to create coroutines within a task and to communicate information among the coroutines. Concurrency mechanisms are provided to create tasks, to synchronize execution of the tasks, and to communicate information between synchronized tasks. When shared memory exists between UNIX processes, UNIX processes are used as virtual processors and task execution is uniformly distributed across them. A clustering mechanism exists to group virtual processors and tasks together, restricting execution of these tasks to only these virtual processors. Partitioning into clusters must be used with care as it has the potential to

*UNIX is a registered trademark of AT&T Bell Laboratories

0038–0644/90/090929–35$17.50
© 1990 by John Wiley & Sons, Ltd.

Received 18 September 1989
Revised 26 March 1990

built. We could further modify the present implementation and eliminate splaying during all insertions and deletions. If deletions/insertions are carried out without splaying then the cost of these operations in a splay tree would be the same as that for a random tree. This modification should help significantly in a dynamic environment where insertions/deletions are occurring frequently.

CONCLUSIONS

We have evaluated the performance of three self-adjusting binary search tree structures and have compared their performance with that of the random BST and the AVL tree. We have found that the search performance of the AVL tree is almost always better than that of any of the self-adjusting BSTs even when the skew factor is close to 1. Of the self-adjusting techniques examined, top-down splaying is easily the best in most situations. Self-adjusting trees perform best in a highly-dynamic environment—which is counter-intuitive. One would expect the best performance to be obtained with a static key set where the trees could adjust properly to the key-access probabilities.

ACKNOWLEDGEMENTS

An earlier version of this paper was presented at the 14th Australian Computer Science Conference held at the University of New South Wales in February 1991. We would like to thank Dr Alistair Moffat of the Department of Computer Science, University of Melbourne for a very careful reading of an earlier version of this paper. His insightful comments have improved this paper considerably.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addision-Wesley 1973.
2. J. Nievergelt and E. M. Reingold, 'Binary search trees of bounded balance', *SIAM J. Computing*, **2**, 33–43 (1973).
3. J. L. Baer, 'Weight-balanced trees', *Proceedings AFIPS 1975 NCC*, **44**, 467–472 (1975).
4. H. Chang and S. S. Iyengar, 'Efficient algorithms to globally balance a binary search tree', *Comm. ACM*, **27**, 695–702 (1984).
5. Q. F. Stout and B. L. Warren, 'Tree rebalancing in optimal time and space', *Comm. ACM*, **29**, 902–908 (1986).
6. E. M. Reingold and W. J. Hansen, *Data Structures*, Little Brown and Company, Boston, 1983.
7. B. Allen and and I. Munro, 'Self-organising binary search trees', *JACM*, **25**, 526–535 (1978).
8. J. R. Bitner, 'Heuristics that dynamically organise data structures', *SIAM J. Computing*, **8**, 82–110 (1979).
9. D. D. Sleator and R. E. Tarjan, 'Self-adjusting binary search trees', *JACM*, **32**, 652–686 (1985).

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(4), 383–404 (APRIL 1993)

Calendrical Calculations, II: Three Historical Calendars

EDWARD M. REINGOLD AND NACHUM DERSHOWITZ
Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, IL 61801–2987, U.S.A.

AND

STEWART M. CLAMEN
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213–3891, U.S.A.

SUMMARY

Algorithmic presentations are given for three calendars of historical interest, the Mayan, French Revolutionary, and Old Hindu.

KEY WORDS Calendar Mayan calendar French Revolutionary calendar Hindu calendar

From a chronological point of view the substitution for the mean calendaric system of one based on the true movements of the sun and moon, was anything but an improvement, as it destabilized the foundations of the time reckoning. Indeed, the system may have had the charm of adapting daily life as nearly as the astronomical knowledge permitted to the movement of the heavenly bodies, but on the other hand it broke ties with history, as there was no unity either of elements or systems. The very complexity of the system is a proof of its primitiveness.

—W. E. van Wijk (1938)

INTRODUCTION

In an earlier paper,[1] the first two authors gave algorithmic descriptions of five calendars in widespread use, namely the Gregorian (New Style), Julian (Old Style), Islamic (Moslem), Hebrew (Jewish), and ISO (International Organization for Standardization) calendars. (For an ancient comparison of the Islamic, Hebrew, and Julian calendars, see al-Biruni's classic treatise.[2]) In this paper, we treat similarly three calendars of historical interest: the Mayan, French Revolutionary (*le calendrier républicain*), and Old Hindu calendars. We follow the same conventions as in our

Received 6 April 1992
Revised 26 October 1992

0038–0644/93/040383–22$16.00
© 1993 by John Wiley & Sons, Ltd.

earlier paper, presenting all algorithms in the form of COMMON LISP[3] functions.* A brief explanation of Lisp, and why we chose it as a vehicle, was given in our earlier paper. As in that paper, we have chosen not to optimize the code at the expense of algorithmic clarity; consequently, improvements are possible in economy and in more appropriate use of Lisp functions.

Aside from their obvious historical value, the three calendars presented here are of interest in that they introduce algorithmic issues not present in the calendars of our previous paper. These new issues are applicable to a wide range of ancient and current calendars. There is also, certainly, an element of whimsy in our selection of ethnically, geographically, and chronologically diverse systems of chrononomy.

As described in our earlier paper, we have chosen Monday, January 1, 1 C.E.† (Gregorian) as our absolute day 1 and count forward day-by-day from there, taking the passage of time as a sequence of days numbered 1, 2, 3, ... that the various human-oriented calendars label differently. For example, absolute day 710,347 is called November 12, 1945 C.E. on the Gregorian calendar. All that is required for calendrical conversion is the ability to convert to and from this absolute calendar.

Our earlier paper gave Lisp functions to do the conversions for the Gregorian, ISO, Julian, Islamic, and Hebrew calendars. For example, to convert an absolute date to a Gregorian date we sequentially determine the year, month, and day of the month—the year is first closely approximated from below and then found precisely by stepping through subsequent years (that is, by a linear search).‡ The month is then found by a similar linear process, and the day of the month is determined by subtraction. The linear searches use a macro (from our previous paper) to compute sums. The expression

---

*To insure correctness, all code in this paper was typeset directly from working Lisp functions. We will gladly provide these Lisp functions in electronic form: send an empty electronic mail message to reingold@cs.uiuc.edu with the subject line containing precisely the phrase 'send-cal'; your message will be answered automatically with the combined code from this paper and our earlier one.

†Common era; or, A.D.

‡In our previous paper we noted that the exact determination of the Gregorian year from the absolute *date* can be viewed as an exercise in base conversion in a mixed-radix system, but (as pointed out to us by Jamshid Afshar), we gave an erroneous description of the conversion. Here is the correct description:

d_0	$= date - 1$	{prior days}
n_{400}	$= d_0$ **div** 146097	{number of completed 400 year cycles}
d_1	$= d_0$ **mod** 146097	{prior days not included in n_{400}}
n_{100}	$= d_1$ **div** 36524	{number of 100 year cycles not included in n_{400}}
d_2	$= d_1$ **mod** 36524	{prior days not included in n_{400} or n_{100}}
n_4	$= d_2$ **div** 1461	{number of 4 year cycles not included in n_{400} or n_{100}}
d_3	$= d_2$ **mod** 1461	{prior days not included in n_{400}, n_{100}, or n_4}
n_1	$= d_3$ **div** 365	{number of years not included in n_{400}, n_{100}, or n_4}
d_4	$= d_3$ **mod** 365	{prior days not included in n_{400}, n_{100}, n_4, or n_1}

if $(n_{100} = 4)$ **or** $(n_1 = 4)$ **then**
  date is **ordinal** day 366 in Gregorian year $400n_{400} + 100n_{100} + 4n_4 + n_1$
else
  date is **ordinal** day $d_4 + 1$ in Gregorian year $400n_{400} + 100n_{100} + 4n_4 + n_1 + 1$.

```
(sum f i k p)
```

computes $\sum_{i \geq k, p(i)} f(i)$; that is, the expression $f(i)$ is summed for all $i = k, k+1, \ldots$, continuing only as long as the condition $p(i)$ holds. The COMMON LISP definition of sum is

```
(defmacro sum (expression index initial condition)
;; Sum expression for index = initial and successive integers,
;; as long as condition holds.
  (let* ((temp (gensym)))
    `(do ((,temp 0 (+ ,temp ,expression))
          (,index ,initial (1+ ,index)))
         ((not ,condition) ,temp))))
```

In this paper we give functions for the Mayan, French Revolutionary, and Old Hindu calendars. Together with the functions in our previous paper, these functions enable conversion back and forth between any of the eight calendars. As in our earlier paper, the algorithms given in this paper do *not* generally work for non-positive absolute dates.

To simplify matters in our earlier paper, we altogether avoided dealing with dates before the common era. For example, our Lisp functions for the Hebrew calendar included the constant 1,373,429 as the number of days on the Hebrew calendar before our absolute date 0. This circumlocution avoided a statement such as 'the epoch of the Hebrew calendar is Saturday, September 5, −3760 (Gregorian)'. In the present paper, however, we deal with two calendars for which the scholarly literature is replete with such statements; thus, to aid the reader interested enough to pursue matters through further reading, we now explain how years before the common era are conventionally handled. This convention is often a source of confusion.

It is computationally convenient, and mathematically sensible, to label years with the sequence of integers $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ so that year 0 precedes year 1; we do this when extrapolating backward on the Gregorian calendar and the same leap-year rule applies based on divisibility by 4, 100, and 400. However, on the Julian calendar it is customary to refer to the year preceding 1 C.E. as 1 B.C.E.,* counting it as a leap year in accordance with the every-fourth-year leap-year rule of the Julian calendar. Thus the epoch of the Hebrew calendar can alternatively be referred to as Saturday, October 5, 3761 B.C.E. (Julian). To highlight this asymmetry, in this paper we will append 'B.C.E.' *only* to Julian calendar years, reserving the minus sign for Gregorian calendar years. Thus for $n \geq 0$,

$$-n \text{ (Gregorian)} = (n+1) \text{ B.C.E. (Julian)},$$

and, for $n \geq 1$,

$$n \text{ (Gregorian)} = n \text{ C.E. (Gregorian)} = n \text{ C.E. (Julian)}.$$

*Before the common era; or, B.C.

Astronomers avoid this confusing situation by specifying dates in 'julian days'.[4, 5] Introduced in 1583 by Joseph Justus Scaliger, this method of dating is a strict counting of days backward and forward from

J.D. 0 = Monday, January 1, 4713 B.C.E. (Julian)
= Monday, November 24, −4711 (Gregorian).

Appendix A, section 1 of Neugebauer[4] explains the choice of starting date in detail. Using Scaliger's system, for example, the epoch of the Hebrew calendar is J.D. 347,996; the literature on the Mayan calendar commonly specifies epochs in julian days. Since our absolute date 0 is J.D. 1,721,425, it follows that

J.D. n = our absolute date $(n - 1721425)$.

THE MAYAN CALENDAR

The Maya, developers of an ancient Amerindian civilization in Central America, employed three separate, overlapping calendrical systems, the *long count*, the *haab*, and the *tzolkin*.[6–13] Their civilization reached its zenith during the period 900–250 B.C.E., and although the Maya survive to this day in Guatemala and in the Yucatan peninsula of Mexico, the precise rules governing their calendars have been lost. What is known today has been recovered through 'astro-archeological' and historical research. There is general agreement on the Mayan calendrical rules; however, the exact correspondence between the Mayan calendars and Western calendars is still a matter of some dispute. Correspondences are proposed by interpreting Mayan recordings of astronomical phenomena, such as eclipses. In this paper, we give the details for the two most popular of the correspondences, the *Goodman–Martinez–Thompson correlation*[6] and *Spinden's correlation*.[11–13*] A superb discussion of Mayan mathematics, astronomy, and calendrical matters is given by Lounsbury.[14]

The long count is a strict counting of days from the beginning of the current cycle, each cycle containing 2,880,000 days (about 7885 solar years); the Maya believed that the universe is destroyed and re-created at the start of every cycle. The units of the long count are

1 kin	= 1 day†	
1 uinal	= 20 kin	(20 days)
1 tun	= 18 uinal	(360 days)
1 katun	= 20 tun	(7200 days)
1 baktun	= 20 katun	(144,000 days)

*Some of Spinden's date calculations are wrong. Here are three examples: on page 46 of 'Maya dates and what they reveal',[13] he gives the equivalence J.D. 1,785,384 = February 10, 176 (Gregorian), but it should be February 11, 176 (Gregorian); on top of page 55 several Gregorian dates are off by one day; on page 57 he gives the equivalence J.D. 2,104,772 = August 30, 1050 (Gregorian), but it should be July 27, 1050 (Gregorian).

†It is uncertain whether the Maya considered the day as beginning at sunset or midnight.

Thus, the long count date 12.16.11.16.6 means 12 baktun, 16 katun, 11 tun, 16 uinal, and 6 kin, for a total of 1,847,486 days from the start of the Mayan calendar epoch. We write Mayan long count dates as Lisp lists of the constituent units, for example, (12 16 11 16 6).

Although not relevant here, the Maya used the following larger units for longer time periods:

1 pictun	= 20 baktun	(2,880,000 days)
1 calabtun	= 20 pictun	(57,600,000 days)
1 kinchiltun	= 20 calabtun	(1,152,000,000 days)
1 alautun	= 20 kinchiltun	(23,040,000,000 days)

An alautun is about 63,081,377 solar years!

The starting epoch of the long count, according to the Goodman–Martinez–Thompson correlation,[6] is taken as Wednesday, August 13, −3113 (Gregorian). This date equals September 8, 3114 B.C.E. (Julian), which equals J.D. 584,285, that is, day −1,137,140 in our absolute reckoning. In other words, our absolute date 0 is long count 7.17.18.13.0, which is 1,137,140 days after the Mayan calendar epoch. Since this value will be used several times, we define it as a Lisp constant,

```
(defconstant mayan-days-before-absolute-zero
;; Number of days of the Mayan calendar epoch before absolute day 0,
;; according to the Goodman-Martinez-Thompson correlation.
  1137140)
```

Spinden[11–13] proposes Monday, October 15, −3373 (Gregorian) as the epoch of the Mayan long count. This date equals November 11, 3374 B.C.E. (Julian), which equals J.D. 489,384, that is, day −1,232,041 in our absolute reckoning. By Spinden's epoch, our absolute date 0 is long count 8.11.2.6.1. To use Spinden's correlation we would instead define

```
(defconstant mayan-days-before-absolute-zero
;; Number of days of the Mayan calendar epoch before absolute day 0,
;; according to the Spinden correlation.
  1232041)
```

Thus, to convert from a Mayan long count date to an absolute date we need only compute the total number of days given by the long count and subtract the number of days before absolute date 0:

```
(defun absolute-from-mayan-long-count (count)
;; Absolute date corresponding to the Mayan long count count,
;; which is a list (baktun katun tun uinal kin).
  (+ (* (first count)  144000);; Baktun.
     (* (second count) 7200)  ;; Katun.
     (* (third count) -360)   ;; Tun.
     (* (fourth count) 20)    ;; Uinal.
     (fifth count)            ;; Kin (days).
     (-                       ;; Days before absolute date 0.
      mayan-days-before-absolute-zero)))
```

The COMMON LISP functions `first`, `second`, `third`, `fourth`, and `fifth` select, respectively, the first through fifth elements of a list.

In the opposite direction, converting an absolute date to a Mayan long count date, we need to add the number of days in the long count before absolute date 0, and then divide the result into baktun, katun, tun, uinal, and kin:

```
(defun mayan-long-count-from-absolute (date)
;; Mayan long count date of absolute date date.
  (let* ((long-count (+ date mayan-days-before-absolute-zero))
         (baktun (quotient long-count 144000))
         (day-of-baktun (mod long-count 144000))
         (katun (quotient day-of-baktun 7200))
         (day-of-katun (mod day-of-baktun 7200))
         (tun (quotient day-of-katun 360))
         (day-of-tun (mod day-of-katun 360))
         (uinal (quotient day-of-tun 20))
         (kin (mod day-of-tun 20)))
    (list baktun katun tun uinal kin)))
```

Here we have used `quotient`, a more aptly named call to COMMON LISP's two-argument version of `floor`,

```
(defun quotient (m n)
   (floor m n))
```

which returns the value $\lfloor m/n \rfloor$.

The Maya used a civil calendar, the haab, based approximately on the solar year, consisting of 18 'months' of 20 days each, together with 5 'monthless' days at the end. The months were called*

(1) Pop	(7) Yaxkin	(13) Mac
(2) Uo	(8) Mol	(14) Kankin
(3) Zip	(9) Chen	(15) Muan
(4) Zotz	(10) Yax	(16) Pax
(5) Tzec	(11) Zac	(17) Kayab
(6) Xul	(12) Ceh	(18) Cumku

The five monthless days were an unlucky period called *Uayeb*. Unlike Gregorian months, the days of the haab months begin at 0 and indicate the number of *elapsed days* in the current month. Thus, 0 Uo follows 19 Pop, and the fifth monthless day is followed by 0 Pop. This method of counting is used for years in the Hindu calendar discussed later in this paper.

Because the haab calendar accounts for only 365 days (as compared to the mean length of the solar tropical year, 365.2422 days), it is assumed the civil calendar slowly drifted with respect to the seasons. Mayan astronomers were aware of this

*The haab and tzolkin month names are transliterated from the Yucatan Mayan dialect. The Guatemalan Maya used slightly different names.

drift, having calculated the solar year to be 365.2420 days long.[15] (The Maya also took careful measurements of the lunar synodic month, which they calculated, quite accurately, to be equal to 29.53059 days.)

The long count date 0.0.0.0.0 is considered to be haab date 8 Cumku (there is no disagreement here in the various correlations), which we specify by

```
(defconstant mayan-haab-at-epoch '(8 18))
```

representing haab dates as pairs `(day month)`, where `day` and `month` are integers in the ranges 0 to 19 and 1 to 19, respectively. Thus we treat Uayeb as a defective nineteenth month and can convert an absolute date to a haab date by

```
(defun mayan-haab-from-absolute (date)
;; Mayan haab date of absolute date date.
  (let* ((long-count (+ date mayan-days-before-absolute-zero))
         (day-of-haab
          (mod (+ long-count
                  (first mayan-haab-at-epoch)
                  (* 20 (1- (second mayan-haab-at-epoch))))
               365))
         (day (mod day-of-haab 20))
         (month (1+ (quotient day-of-haab 20))))
    (list day month)))
```

It is not possible to convert a haab date to an absolute date, since, without a 'year', there is no unique corresponding absolute date. We can ask, though, for the number of days between two dates on the haab calendar—the calculation is elementary:

```
(defun mayan-haab-difference (date1 date2)
;; Number of days from Mayan haab date date1 to the next
;; occurrence of Mayan haab date date2.
  (mod (+ (* 20 (- (second date2) (second date1)))
          (- (first date2) (first date1)))
       365))
```

Note that COMMON LISP's `mod`, unlike `rem`, *always returns a non-negative value for a positive divisor*; we use this property in the last function and frequently below. In the absence of such a modulus function, a conditional test could be used to map a remainder of -1 to $n-1$, and so on. The function `mod` also applies to non-integer values, so, for example, `(mod x 1)` returns the fractional part of `x`.

The function `mayan-haab-difference` can be used as follows to compute the absolute date of the Mayan haab date on or before a given absolute date:

```
(defun mayan-haab-on-or-before (haab date)
;; Absolute date of latest date on or before absolute date date
;; that is Mayan haab date haab.
   (- date
      (mod (- date
              (mayan-haab-difference
               (mayan-haab-from-absolute 0) haab))
           365)))
```

The third Mayan calendar, the tzolkin (or divine) calendar, was a religious calendar consisting of two cycles, a thirteen day count and a cycle of twenty names:

(1) Imix	(5) Chicchan	(9) Muluc	(13) Ben	(17) Caban
(2) Ik	(6) Cimi	(10) Oc	(14) Ix	(18) Etznab
(3) Akbal	(7) Manik	(11) Chuen	(15) Men	(19) Cauac
(4) Kan	(8) Lamat	(12) Eb	(16) Cib	(20) Ahau

Unlike the haab, the counts and names cycle *simultaneously*, so, for example, 13 Etznab precedes 1 Cauac which precedes 2 Ahau which precedes 3 Imix, and so on. Since 20 and 13 are relatively prime, this progression results in 260 unique dates, forming the divine year.

For the tzolkin calculations, and in the remainder of the paper, we will need a function like mod, but with its values adjusted so that the modulus of a multiple of the divisor is the divisor itself, rather than 0:

```
(defun adjusted-mod (m n)
;; Positive remainder of m/n with n instead of 0.
  (1+ (mod (1- m) n)))
```

The long count date 0.0.0.0.0 is taken to be tzolkin date 4 Ahau (the different correlations agree on this, too). Representing tzolkin dates as pairs of positive integers (number name), where number and name are integers in the ranges 1 to 13 and 1 to 20, respectively, we specify

```
(defconstant mayan-tzolkin-at-epoch '(4 20))
```

As with the haab, we can convert from an absolute date to a tzolkin date with

```
(defun mayan-tzolkin-from-absolute (date)
;; Mayan tzolkin date of absolute date date.
  (let* ((long-count (+ date mayan-days-before-absolute-zero))
         (number
          (adjusted-mod (+ long-count
                           (first mayan-tzolkin-at-epoch))
                        13))
         (name
          (adjusted-mod (+ long-count
                           (second mayan-tzolkin-at-epoch))
                        20)))
    (list number name)))
```

Just as with the haab calendar, it is impossible to convert a tzolkin date to an absolute date. Unlike the haab calendar, however, because day numbers and day names cycle simultaneously, to calculate the number of days between two given tzolkin dates requires the solution to a pair of simultaneous linear congruences.[16] Suppose we want to know the number of days x from tzolkin date $(d_1\ n_1)$ until the next occurrence of tzolkin date $(d_2\ n_2)$. We must have

$$d_1 + x \equiv d_2 \pmod{13},$$

or, equivalently, $x = d_2 - d_1 + 13i$, for some integer i. Similarly, we must have

$$x \equiv n_2 - n_1 \pmod{20},$$

which becomes

$$d_2 - d_1 + 13i \equiv n_2 - n_1 \pmod{20}.$$

Hence we need to know the values of i satisfying

$$13i \equiv n_2 - n_1 - d_2 + d_1 \pmod{20}.$$

Multiplying each side by -3, the multiplicative inverse of 13 modulo 20, gives

$$i \equiv 3(d_2 - d_1 - n_2 + n_1) \pmod{20},$$

from which we conclude that

$$x = d_2 - d_1 + 13[3(d_2 - d_1 - n_2 + n_1) \bmod 20].$$

However, because we want the *next* occurrence of $(d_2\ n_2)$, we must guarantee that x is non-negative. Thus we write,

```
(defun mayan-tzolkin-difference (date1 date2)
;; Number of days from Mayan tzolkin date date1 to the next
;; occurrence of Mayan tzolkin date date2.
  (let* ((number-difference (- (first date2) (first date1)))
         (name-difference (- (second date2) (second date1))))
    (mod (+ number-difference
            (* 13 (mod (* 3 (- number-difference name-difference))
                       20)))
         260)))
```

As with the haab calendar, this function can be used to compute the absolute date of the Mayan tzolkin date on or before a given absolute date:

```
(defun mayan-tzolkin-on-or-before (tzolkin date)
;; Absolute date of latest date on or before absolute date date
;; that is Mayan tzolkin date tzolkin.
    (- date
       (mod (- date (mayan-tzolkin-difference
                     (mayan-tzolkin-from-absolute 0)
                     tzolkin))
            260)))
```

A popular way for the Maya to specify a date was to use the haab and tzolkin dates together, forming a cycle of the least common multiple of 365 and 260 days, 18980 days or approximately 52 solar years. This cycle is called a *calendar round*,[15, 17] and we can ask for the most recent absolute date, on or before a given absolute date, that falls on a specified date of the calendar round.

Suppose the haab date of interest is $(D\ M)$ and the tzolkin date of interest is $(d\ n)$; we seek the latest absolute date x, on or before the given absolute date, that satisfies

$$\begin{aligned} x + D_0 &\equiv D \pmod{20}, \\ x + M_0 &\equiv M \pmod{18}, \\ x + d_0 &\equiv d \pmod{13}, \\ x + n_0 &\equiv n \pmod{20}, \end{aligned}$$

where $(D_0\ M_0)$ is the haab date of absolute date 0, and $(d_0\ n_0)$ is the tzolkin date of absolute date 0. The first two of these congruences combine to become

$$x \equiv \Delta_H \pmod{365}, \tag{1}$$

where

$$\Delta_H = \texttt{mayan-haab-difference} \text{ applied to } (D\ M) \text{ and } (D_0\ M_0).$$

The last two of these congruences combine to become

$$x \equiv \Delta_T \pmod{260}, \tag{2}$$

where

$$\Delta_T = \texttt{mayan-tzolkin-difference} \text{ applied to } (d\ n) \text{ and } (d_0\ n_0).$$

Congruence (1) means that

$$x = \Delta_H + 365i, \tag{3}$$

and combining this with congruence (2) we get

$$365i \equiv \Delta_T - \Delta_H \pmod{260}.$$

This has a solution *only* if $\Delta_T - \Delta_H$ is divisible by 5; otherwise the haab-tzolkin combination cannot occur. So, dividing by 5 we get

$$73i \equiv \frac{\Delta_T - \Delta_H}{5} \pmod{52},$$

and multiplying this by the multiplicative inverse of 73 modulo 52 (which, by coincidence, is 5), we find

$$i \equiv \Delta_T - \Delta_H \pmod{52},$$

so that $i = \Delta_T - \Delta_H + 52u$, for some u. Plugging this into equation (3) yields

$$x = \Delta_H + 365(\Delta_T - \Delta_H) + 18980u.$$

Thus we want the last date on or before the given absolute date that is congruent to $\Delta_H + 365(\Delta_T - \Delta_H)$ modulo 18980. This is computed in a manner identical to that in the function `Kday-on-or-before` in our earlier paper,[1] and we have:

```
(defun mayan-haab-tzolkin-on-or-before (haab tzolkin date)
;; Absolute date of latest date on or before date that is Mayan
;; haab date haab and tzolkin date tzolkin; returns nil if such
;; a haab-tzolkin combination is impossible.
  (let* ((haab-difference
          (mayan-haab-difference (mayan-haab-from-absolute 0)
                                 haab))
         (tzolkin-difference
          (mayan-tzolkin-difference (mayan-tzolkin-from-absolute 0)
                                    tzolkin))
         (difference (- tzolkin-difference haab-difference)))
    (if (= (mod difference 5) 0)
        (- date
           (mod (- date
                   (+ haab-difference (* 365 difference)))
                18980))
      nil)));;  haab-tzolkin combination is impossible.
```

This function can be used to compute the number of days between a pair of dates on the calendar round or to write a function `mayan-haab-tzolkin-on-or-after`; we leave these to the reader.

THE FRENCH REVOLUTIONARY CALENDAR

The French Revolutionary Calendar[9, 18] was instituted by the National Convention of the French Republic in October 1793. Its epoch was absolute date 654,415, that is, Saturday, September 22, 1792 (Gregorian), the autumnal equinox of that year, and also the first day following the establishment of the Republic. It went into effect on Sunday, November 24, 1793 (Gregorian), and was used by the French until Tuesday, December 31, 1805 (Gregorian); on Wednesday, January 1, 1806 (Gregorian), the Revolutionary calendar was abandoned by Napoleonic edict and France reverted to the Gregorian calendar.

Following the example of several ancient calendars, including the Coptic, Ethiopic, and Persian,[9] the French Revolutionary calendar divided the year into 12 months containing exactly 30 days each, followed by a period of five monthless days (six in leap years). The names of the twelve months, poetically coined by Fabre d'Eglantine, were taken from the seasons in which they occurred:

(1) Vendémiaire (vintage)
(2) Brumaire (fog)
(3) Frimaire (sleet)
(4) Nivôse (snow)
(5) Pluviôse (rain)
(6) Ventôse (wind)
(7) Germinal (seed)
(8) Floréal (blossom)
(9) Prairial (pasture)
(10) Messidor (harvest)
(11) Thermidor (heat)
(12) Fructidor (fruit)

Contemporary British journalists sarcastically dubbed them Slippy, Nippy, Drippy, Freezy, Wheezy, Sneezy, Showery, Flowery, Bowery, Wheaty, Heaty, Sweety.

We will use a list `(month day year)` to represent the date, treating the monthless days as a thirteenth month, as in the Mayan haab calendar:

```
(defun french-last-day-of-month (month year)
;; Last day of  month, year on the French Revolutionary calendar.
  (if (< month 13)
      30
    (if (french-leap-year year)
        6
      5)))
```

The leap-year rules are given below.

Although not relevant to our calculations, each month was divided into three 'decades' of ten days each; the tenth day was considered a day of rest. (This made the new calendar unpopular because under the Gregorian calendar the workers had had every seventh day off.) The ten days were named by their ordinal position in the decade,

(1) Primidi	(6) Sextidi
(2) Doudi	(7) Septidi
(3) Tridi	(8) Oxtidi
(4) Quartidi	(9) Nonidi
(5) Quintidi	(10) Decadi

Each day was divided into ten 'hours', each of which was divided into one hundred 'minutes', each of which was divided into one hundred 'seconds'.

The five or six monthless days that were added at the end of each year were holidays called *sansculottides*, celebrating various attributes of the Revolution:

(1) Jour de la Vertu (virtue day)
(2) Jour du Genie (genius day)
(3) Jour du Labour (labor day)
(4) Jour de la Raison (reason day)
(5) Jour de la Recompense (reward day)
{(6) Jour de la Revolution (revolution day)}

The leap-year structure is given in curly brackets.

The leap year rule is complicated by historical fact.[18] Originally, the calendar was to be kept in synchronization with the solar year by using the extra day in a leap year to force the new year, 1 Vendémiaire, to occur at the autumnal equinox. Unfortunately, the irregular variations of a day or two in the occurrence of the equinox made this impracticable and a proposal was put forward for a simpler, more regular, leap year rule; however, the calendar was abandoned before this rule could be adopted. Thus, leap years *actually occurred* in the 3rd, 7th, and 11th years and *would have occurred* in the 15th and 20th years, had the calendar still been in use. The proposed rule then would have been:

every 4th year is a leap year, except
every 100th year is not a leap year, except
every 400th year is a leap year, except
every 4000th year is not a leap year,

giving an average of 365.24225 days per year, an error of about 1 day in 20,000 years.

The following Lisp function uses historical practice based on equinoxes for years 1 through 20 and uses the proposed rule for years beyond 20:

```
(defun french-leap-year (year)
;; True if year is a leap year on the French Revolutionary calendar.
  (or (member year '(3 7 11));; Actual.
      (member year '(15 20)) ;; Anticipated.
      (and (> year 20)       ;; Proposed.
           (= 0 (mod year 4))
           (not (member (mod year 400) '(100 200 300)))
           (not (= 0 (mod year 4000))))))
```

Conversion of a French Revolutionary date to an absolute date is thus done by summing all days before that date, including the number of days before the calendar began, 365 days for each prior year, all prior leap days, and the number of prior days in the present year:

```
(defun absolute-from-french (date)
;; Absolute date of French Revolutionary date.
  (let* ((month (first date))
         (day (second date))
         (year (third date)))
    (+ 654414;; Days before start of calendar.
       (* 365 (1- year));; Days in prior years.
       ;; Leap days in prior years.
       (if (< year 20)
           (quotient year 4);; Actual and anticipated practice,
                            ;; that is, years 3, 7, 11, and 15.
         ;; Proposed rule--there were 4 leap years before year 20.
         (+ (quotient (1- year) 4)
            (- (quotient (1- year) 100))
            (quotient (1- year) 400)
            (- (quotient (1- year) 4000))))
       (* 30 (1- month));; Days in prior months this year.
       day)));; Days so far this month.
```

Calculating the French Revolutionary date from the absolute date d involves sequentially determining the year, month, and day of the month. The year is first closely approximated from below by $\lfloor d/366 \rfloor$ and then found precisely by stepping through subsequent years (that is, by a linear search). The month is then found by a similar linear process, and the day of the month is determined by subtraction:

```
(defun french-from-absolute (date)
;; French Revolutionary date (month day year) of absolute date;
;; returns nil if date is before the French Revolution.
  (if (< date 654415)
      nil;; pre-French Revolutionary date.
    (let* ((approx       ;; Approximate year from below.
```

```
    (quotient (- date 654414) 366))
   (year          ;; Search forward from the approximation.
    (+ approx
       (sum 1 y approx
         (>= date
             (absolute-from-french (list 1 1 (1+ y)))))))
   (month         ;; Search forward from Vendemiaire.
    (1+ (sum 1 m 1
          (> date
             (absolute-from-french
              (list m
                    (french-last-day-of-month m year)
                    year))))))
   (day           ;; Calculate the day by subtraction.
    (- date
       (1- (absolute-from-french (list month 1 year))))))
(list month day year))))
```

THE OLD HINDU CALENDARS

The Hindus have both solar and lunar calendars. Solar dates are in use primarily in southern India and lunar dates are used elsewhere; virtually all religious holidays are determined by the lunar calendar. In the Hindu lunar system, like other lunisolar calendars, months follow the lunar cycle and are synchronized with the solar year by introducing occasional leap months. Unlike the lunisolar Hebrew calendar (described in algorithmic detail in our previous paper[1]), the Hindu intercalated months do not follow a fixed pattern; instead, like the lunisolar Chinese calendar,[19] their occurrence depends on astronomical factors. However, unlike other calendars, a day can be *omitted* any time in the lunar month.

Current Hindu calendars are based on approximations to the *true* times of the sun's entrance into the signs of the zodiac and of lunar conjunctions (new moons). But, some earlier Hindu calendars appear instead to have used an approximation to *mean* times.[20, 21] It is this mean calendar, as described in sections 6–15 of van Wijk,[20] that we implement here. The mean and true calendars can differ by a few days and/or can be shifted by a month. For an ancient description of Hindu astronomy, calendars, and holidays, see the book on India by al-Biruni.[22*]

There are various epochs that are, or have been, used as starting points for the enumeration of years in India, and for each there are two versions, one beginning with year 0 (expired years) and the other with year 1 (current year). We use the *Kali Yuga* ('Iron Age') epoch, according to which the beginning of the first day of year 0 K.Y. is midnight at the start of Friday, January 23, −3101 (Gregorian). This

*There is some confusion of dates in the note (on page 358) by R. Schram to volume II, page 2 of Sachau's translation of this book, where the following equivalences are given: Thursday, February 25, 1031 C.E. (Julian) = 1 Caitra 953 Śaka Era = 28 Ṣafar 422 Anno Hijrae = 19 Ispandârmadh-Mâh 399 Anno Persarum, and New Year 400 Anno Persarum = March 9, 1031 C.E. (Julian) = J.D. 2,097,686. In fact, February 25, 1031 C.E. (Julian) = 29 Ṣafar 422 Anno Hijrae = J.D. 2,097,686.

date equals February 18, 3102 B.C.E. (Julian), which equals J.D. 588,466, that is, day $-1,132,959$ in our absolute reckoning. That midnight is considered to have been the start of a new solar year and a new lunar month; indeed, according to the data in the *Sūrya-Siddhānta*,[23] it is the time of the most recent conjunction of all the visible planets. The (expired) year number is the number of solar (sidereal, not tropical*) years that have *elapsed* since the onset of the Kali Yuga. As van Wijk[20] explains:

> We count the years of human life in expired years. A child of seven years has already lived more than seven years; but on the famous *18 Brumaire de l'An VIII de la République Française une et indivisible* only 7 years and 47 days of the French Era had elapsed.

The Kali Yuga epoch marks—in Hindu chronology—the onset of the fourth and final stage (lasting 432,000 years) of the 4,320,000-year era beginning with the last re-creation of the world.

The solar months are named after the signs of the zodiac (but may differ from region to region):

(1) Mesha	(Aries)	(7) Tulā	(Libra)
(2) Vṛshabha	(Taurus)	(8) Vṛiśchika	(Scorpio)
(3) Mithuna	(Gemini)	(9) Dhanus	(Sagittarius)
(4) Karka	(Cancer)	(10) Makara	(Capricorn)
(5) Simha	(Leo)	(11) Kumbha	(Aquarius)
(6) Kanyā	(Virgo)	(12) Mīna	(Pisces)

The (sidereal) solar year is taken to be

$$\frac{1,577,917,828}{4,320,000} = 365.25875648148148\cdots$$

(civil) days long. Different Indian astronomical treatises give slightly varying constants; this, and the constants that follow, are from the *Sūrya-Siddhānta*,[23] circa 1000 C.E. The solar new year is called *Mesha saṃkrānti*; it is the day of the first sunrise after the sun enters Mesha. Each subsequent month begins when the sun has traveled one twelfth of the way around the ecliptic (the sun's path in the sky) and entered a new sign of the zodiac. If the sign is entered before sunrise, then it is day one of a new month; otherwise, it is the last day of the previous month. Hence, even though in the mean system the speed of the sun is taken to be constant, months vary in length.

The name of a lunar month depends on the position of the sun in the zodiac at the beginning of that lunar month, and is derived from the names of asterisms (star groups) along the ecliptic:

---

*A *tropical year* is the time it takes for the sun to return to the same position in its apparent path, while a *sidereal year* is the time it takes for the sun to return to the same celestial longitude; the difference is slight.

(1) Chaitra
(2) Vaiśākha
(3) Jyaishṭha
(4) Āshāḍha
(5) Śrāvaṇa
(6) Bhādrapada
(7) Āśvina
(8) Kārttika
(9) Mārgaśīra
(10) Pausha
(11) Māgha
(12) Phālguna

Some regions of India begin the year with Kārttika and/or use different or shifted month names. We follow the south-India method in which months begin and end at new moons (the *amânta* scheme); in the north, months go from full moon to full moon (the *pûrṇimânta* scheme). Our absolute day 0 is 18 Makara, 3101 K.Y. on the mean solar calendar and 19 Pausha on the lunar one.

Since a solar month (the mean time for the sun to traverse a zodiacal sign) is longer than a lunar month, a lunar month is intercalated whenever two new moons occur within one sign of the Hindu zodiac. The two months starting within that sign take the same name, except that the first is called *adhika* ('added') and the second *nija* ('regular').

The lunar cycle is divided into thirty phases (or 'lunar days'), called *tithis*; because a mean lunar month is less than thirty (civil) days long, a tithi is somewhat shorter than a day. The first fifteen tithis belong to the *suddha* ('bright', waxing) fortnight and the second fifteen, to the *bahula* ('dark', waning) fortnight. The day of the lunar month corresponds to the phase of the moon at sunrise, and is usually referred to by ordinal number within one fortnight or the other; we use ordinal numbers from 1 to 30. Just as there are leap months, caused by two new moons occurring within the same zodiacal sign, there are also 'lost' days whenever a tithi begins and ends between one sunrise and the next.

Suppose we can determine the longitudinal positions of the sun and moon, relative to the celestial sphere, at any given time. Then to determine the lunar Hindu date of any given day, we perform the following sequence of operations:

1. Determine the phase of the moon at sunrise of the given day, by taking the difference in longitudes between the positions of the sun and moon. This gives the ordinal number of the tithi current at sunrise.

2. Determine when the last new moon was before sunrise of the current day.

3. Determine the position of the sun at that new moon. The zodiacal sign in which it was determines the name of the current month.

4. Compare the current month name with that of the next new moon. If they are the same, then it is a leap month.

The calendars in use in recent times follow a similar pattern, but require much more elaborate calculation to approximate the true positions of the sun and moon and the time of local sunrise. These more elaborate calculations can result, on rare occasions, in expunged months and intercalated days.[20, 24]

Four constants play a role in the mean computations:

```
(defconstant solar-sidereal-year (+ 365 279457/1080000))
(defconstant solar-month (/ solar-sidereal-year 12))
(defconstant lunar-sidereal-month (+ 27 4644439/14438334))
(defconstant lunar-synodic-month (+ 29 7087771/13358334))
```

We work exclusively with rational numbers; otherwise, double precision is required for many of the calculations. Unlike table-based methods, using rational numbers gives perfect fidelity to the sources.

The sidereal year is the mean number of days it takes for the sun to return to the same longitudinal point with respect to the stars, and the solar month is the mean time it takes it to traverse one sign of the zodiac. Similarly, the sidereal month is the mean time it takes for the moon to return to the same point, in which time the sun would have moved farther on. The synodic month takes the motion of the sun into account—it is the mean time between new moons (lunar conjunctions). Civil days begin at sunrise; we use mean sunrise, one quarter of a day past midnight, that is, 6:00A.M., in our calculations.

Once we know the position of the sun in degrees of celestial longitude,

```
(defun solar-longitude (days)
;; Mean sidereal longitude of the sun, in degrees,
;; at date and fraction of day days.
  (* (mod (/ days solar-sidereal-year) 1) 360))
```

we can determine the zodiacal sign:

```
(defun zodiac (days)
;; Zodiacal sign of the sun, as integer in range 1..12,
;; for date and fraction of day days.
  (1+ (quotient (solar-longitude days) 30)))
```

Now, converting an absolute date to the date according to the Hindu solar calendar is straightforward:

```
(defun old-hindu-solar-from-absolute (date)
;; Hindu solar month, day, and year of absolute date date.
  (let* ((hdate (+ date 1132959 1/4));;   Sunrise on Hindu date.
         (year (quotient hdate solar-sidereal-year))
         (month (zodiac hdate))
         (day (1+ (floor (mod hdate solar-month)))))
    (list month day year)))
```

Inverting this calculation is no harder:

```
(defun absolute-from-old-hindu-solar (date)
;; Absolute date corresponding to Hindu solar date date.
  (let* ((month (first date))
         (day (second date))
         (year (third date)))
    (floor (+ (* year solar-sidereal-year);; Days in elapsed years.
              (* (1- month) solar-month)  ;; In months.
              day -1/4                    ;; Whole days until midnight.
              -1132959))))                ;; Days before absolute day 0.
```

To compute the lunar date, we need to determine the position of the moon in degrees of celestial longitude:

```
(defun lunar-longitude (days)
;; Mean sidereal longitude of the moon, in degrees,
;; at date and fraction of day days.
  (* (mod (/ days lunar-sidereal-month) 1) 360))
```

To get the phase of the moon, we must measure the distance between the positions of the sun and the moon. The following function converts that value, in degrees, to a number between 1 and 30:

```
(defun lunar-phase (days)
;; Longitudinal distance between the sun and the moon, as an integer
;; in the range 1..30, at date and fraction of day days.
  (1+ (quotient
        (mod (- (lunar-longitude days) (solar-longitude days))
             360)
        12)))
```

To determine the number of the month, we look back to the last occurrence of a new moon, see where the sun was at that time, and then give the lunar month the number of the next solar month. Similarly, for the lunar year number, we use the solar year in which the next lunar month falls. The most recent new moon is found by

```
(defun new-moon (days)
;; Time of the most recent mean conjunction at or before
;; date and fraction of day days.
  (- days (mod days lunar-synodic-month)))
```

We are now ready to compute the Hindu date, given any absolute date. We use a list `(month leapmonth day year)` to represent a date, where `month` is an integer in the range 1 through 12, `day` is an integer in the range 1 through 30, `year` is a non-negative integer, and `leapmonth` is either `t` or `nil`, corresponding to 'true' and 'false':

```
(defun old-hindu-lunar-from-absolute (date)
;; Hindu lunar month, day, and year of absolute date date.
  (let* ((hdate (+ date 1132959))          ;; Hindu date.
         (sunrise (+ hdate 1/4))           ;; Sunrise on that day.
         (last-new-moon                    ;; Last new moon.
          (new-moon sunrise))
         (next-new-moon                    ;; Next new moon.
          (+ last-new-moon lunar-synodic-month))
         (day (lunar-phase sunrise))       ;; Day of month.
         (month                            ;; Month of lunar year.
          (adjusted-mod (1+ (zodiac last-new-moon)) 12))
         (leapmonth                        ;; If next month the same.
```

```
      (= (zodiac last-new-moon)
         (zodiac next-new-moon)))
     (next-month                        ;; Beginning of next month.
      (+ next-new-moon
                 (if leapmonth lunar-synodic-month 0)))
     (year                              ;; Solar year of next month.
      (quotient next-month solar-sidereal-year)))
   (list month leapmonth day year)))
```

Unlike the French Revolutionary calendar, or the calendars in our previous paper,[1] there is no direct way of converting a lunar Hindu date to an absolute date, since years and months do not have a fixed length. To invert the process and derive the absolute date from a Hindu lunar date, we first find a lower bound on the possible absolute date, and then perform a linear search for the exact correspondence. For this purpose, it is convenient to be able to compare Hindu lunar dates:

```
(defun old-hindu-lunar-precedes (date1 date2)
;; True if Hindu lunar date1 precedes date2.
  (let* ((month1 (first date1))
         (month2 (first date2))
         (leap1 (second date1))
         (leap2 (second date2))
         (day1 (third date1))
         (day2 (third date2))
         (year1 (fourth date1))
         (year2 (fourth date2)))
    (or (< year1 year2)
        (and (= year1 year2)
             (or (< month1 month2)
                 (and (= month1 month2)
                      (or (and leap1 (not leap2))
                          (and (equal leap1 leap2)
                               (< day1 day2)))))))))
```

Finally, taking into account the possible nonexistence of a particular date,

```
(defun absolute-from-old-hindu-lunar (date)
;; Absolute date corresponding to Hindu lunar date date;
;; returns nil if no such date exists.
  (let* ((years (fourth date))      ;; Elapsed years.
         (months (- (first date) 2));; Elapsed whole months,
              ;; minus a month's possible difference between the
              ;; solar and lunar year.
         (approx;;  Approximate date from below by adding days...
          (+ (floor (* years solar-sidereal-year)) ;;  in years,
             (floor (* months lunar-synodic-month));;  in months,
             -1132959))             ;; and before absolute date 0.
         (try
          (+ approx                 ;; Search forward to correct date,
             (sum i 1 i approx      ;; or just past it.
```

```
                (old-hindu-lunar-precedes
                 (old-hindu-lunar-from-absolute i)
                 date)))))
    (if (equal (old-hindu-lunar-from-absolute try) date)
        try
      nil)));; date non-existent on Hindu lunar calendar.
```

CONCLUSION

Our goal in this series of papers is to make accurate calendrical algorithms readily available, since calendrical problems are notorious for plaguing software. Some examples:

1. The COBOL programming language allocates only two decimal digits for internal storage of years, so untold numbers of programs are expected to go awry on New Year's Eve of the coming century.[25]

2. Many programs err in, or simply ignore, the century rule for leap years on the Gregorian calendar (every 4th year is a leap year, except every 100th year which is not, except every 400th year which is). For example, early releases of the popular spreadsheet program Lotus® 1-2-3® treated 2000 as a non-leap year, a problem eventually fixed. But, all releases of Lotus® 1-2-3® take 1900 as a leap year; by the time this error was recognized, the company deemed it too late to correct: 'The decision was made at some point that a change now would disrupt formulas which were written to accomodate [*sic*] this anomaly.'[26]

3. Various programs calculate the Hebrew calendar by first determining the date of Passover using Gauss's method;[27] this method is correct only when sufficient precision is used, so such an approach often leads to errors.

4. At least one modern, standard source for calendrical matters, Parise,[9] has errors, some of which are presumably due to the algorithms used to produce the tables.*

Though the calendars described in this paper are mainly of historical interest, the Mayan and French Revolutionary calendars are incorporated in version 19 of GNU Emacs.[28] The Hindu can serve as the basis for implementing the various more intricate calendars in use on the Indian subcontinent today, and perhaps also the Chinese calendar.[19] The algorithms presented also serve to illustrate some basic features of nonstandard calendars: The Mayan calendar requires dealing with multiple, independent cycles and exemplifies the kind of reasoning often needed for calendrical-historical research. The French calendar is an example of one in which two cycles (years and 'decades') are synchronized. The Hindu calendar is an example of one in which the cycles (days of the month, months of the year) are irregular.

---

*Examples include: The Mayan date 8.1.19.0.0 is given incorrectly as February 14, 80 (Gregorian) on page 290; the dates given on pages 325–327 for Easter for the years 1116, 1152, and 1582 are not Sundays; the epact given for 1986 on page 354 is wrongly given as 20.

ACKNOWLEDGEMENTS

The second author's research was supported in part by the U. S. National Science Foundation under Grants CCR-90-07195 and CCR-90-24271 and by a Lady Davis fellowship at the Hebrew University in Jerusalem.

REFERENCES

Note: Only English language references are included in the following list. Some of these are relatively inaccessible.

1. N. Dershowitz and E. M. Reingold, 'Calendrical calculations', *Software—Practice and Experience*, **20**, (9), 899–928 (Sept. 1990).
2. Al-Biruni (= Abû-Raiḥân Muḥammad ibn 'Aḥmad al-Bîrûnî), *Alâthâr Albâḳiya 'an-il-Ḳûrûn Alkhâliya*, 1000. Translated and annotated by C. E. Sachau as *The Chronology of Ancient Nations*, William H. Allen and Co., London, 1879; reprinted by Hijra International Publishers, Lahore, Pakistan, 1983.
3. G. L. Steele, Jr., COMMON LISP: *The Language*, 2nd edn., Digital Press, Bedford, MA, 1990.
4. O. Neugebauer, *A History of Ancient Mathematical Astronomy*, Springer-Verlag, Berlin, 1975.
5. *Explanatory Supplement to the Astronomical Ephemeris and the American Ephemeris and Nautical Almanac*, Her Majesty's Stationery Office, London, 1961.
6. J. E. S. Thompson, *Maya Hieroglyphic Writing*, 3rd edn., University of Oklahoma Press, Norman, OK, 1971.
7. L. Satterwaite, 'Concepts and structures of Maya calendrical arithmetics', *Ph.D. Thesis*, University of Pennsylvania, Philadelphia, 1947.
8. B. Tedlock, *Time and the Highland Maya*, University of New Mexico Press, Albuquerque, 1982.
9. F. Parise, *The Book of Calendars*, Facts on File, New York, 1982.
10. J. Hastings, ed., *Encyclopædia of Religion and Ethics*, Charles Scribner's Sons, New York, 1911.
11. H. J. Spinden, 'Central American calendars and the Gregorian day', *Proc. Nat. Acad. Sci. (USA)*, **6**, 56–59 (1920).
12. H. J. Spinden, 'The reduction of Maya dates', *Peabody Museum Papers*, **VI**, (4), (1924).
13. H. J. Spinden, 'Maya dates and what they reveal', *Science Bulletin* (The Museum of the Brooklyn Institute of Arts and Sciences), **IV**, (1), (1930).
14. F. G. Lounsbury, 'Maya numeration, computation, and calendrical astronomy', *Dictionary of Scientific Bibliography*, **15**, Supplement 1, 759–818, Charles Scribner's Sons, New York, 1978.
15. S. G. Morley, *The Ancient Maya*, revised by G. W. Brainerd, Stanford University Press, Stanford, CA, 1963.
16. O. Ore, *Number Theory and Its History*, McGraw-Hill Book Co., Inc., New York, 1948. Reprinted by Dover Publications, Inc., Mineola, NY, 1987.
17. C. P. Bowditch, *The Numeration, Calendar Systems and Astronomical Knowledge of the Mayas*, Cambridge University Press, Cambridge, 1910.
18. M. Hamer, 'A calendar for all seasons', *New Scientist*, **124**, (1696/1697), 9–12 (Dec. 23/30, 1989).
19. H. Fritsche, *On Chronology and the Construction of the Calendar with Special Regard to the Chinese Computation of Time Compared with the European*, R. Laverentz, St. Petersburg, 1886.
20. W. E. van Wijk, *Decimal Tables for the Reduction of Hindu Dates from the Data of the Sūrya-Siddhānta*, Martinus Nijhoff, The Hague, 1938.

21. H. Jacobi, 'The computation of Hindu dates in inscriptions, &c.', *Epigraphia Indica: A Collection of Inscriptions Supplementary to the Corpus Inscriptionum Indicarum of the Archæological Survey of India*, J. Burgess, ed., Calcutta, 1892, 403–460, 481.
22. Al-Biruni (= Abû-Raiḥân Muḥammad ibn 'Aḥmad al-Bîrûnî), *India: An accurate description of all categories of Hindu thought, as well those which are admissible as those which must be rejected*, circa 1030. Translated and annotated by E. C. Sachau, *Albêrûnî's India: An account of the religion, philosophy, literature, geography, chronology, astronomy, customs, laws and astrology of India*, William H. Allen and Co., London, 1910; reprinted under the Authority of the Government of West Pakistan, Lahore, 1962 and by S. Chand & Co., New Delhi, 1964.
23. *Sūrya-Siddhānta*, circa 1000. Translated by E. Burgess with notes by W. D. Whitney, *J. Amer. Oriental Soc.*, New Haven, 1860. A new edition, edited by P. Gangooly with an introduction by P. Sengupta, was published by Calcutta University, 1935. This edition was reprinted by Indological Book House, Varanasi, India, 1977.
24. V. B. Ketkar, 'Indian and foreign chronology: with theory, practice and tables, B.C. 3102 to 2100 A.D. and notices of the Vedic, the ancient Indian, the Chinese, the Jewish, the ecclesiastical and the Coptic calendars', *J. Royal Asiatic Soc., Bombay Branch*, **XXVI**, (LXXV-A, *Extra Number*), 1923.
25. P. G. Neumann, 'Inside risks: the clock grows at midnight', *Comm. ACM*, **34**, (1), 170 (Jan., 1991).
26. Letter to Nachum Dershowitz from Kay Wilkins, Customer Relations Representative, Lotus Development Corporation, Cambridge, MA, April 21, 1992.
27. I. Rhodes, 'Computation of the dates of the Hebrew New Year and Passover', *Comp. & Maths. with Appls.*, **3**, 183–190 (1977).
28. R. M. Stallman, *GNU Emacs Manual*, 6th edn., Free Software Foundation, Cambridge, MA, 1986.

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(4), 405–418 (APRIL 1993)

Object Identification in the Lego Kernel

LARRY HUGHES
Department of Mathematics and Computing Science, Saint Mary's University, Halifax, Nova Scotia, B3H 3C3, Canada

SUMMARY

In a message-based operating system, the kernel is responsible for forwarding messages to destination objects using a destination identifier. Part of the kernel overhead in processing the message involves mapping the identifier into a network address. This paper describes how the choice of identifier structure has simplified the techniques used by the Lego kernel when handling messages, notably: minimizing the overheads associated with address mapping, eliminating the reception of unwanted network packets, while permitting both individual (unicast) and group (multicast) communications.

KEY WORDS Operating systems Communications Multicast

INTRODUCTION

The past decade has seen radical changes in the design of operating systems. Operating systems such as Mach,[1], Sprite,[2] and Topaz[3] have evolved away from monolithic structures to much more modular designs.

Central to the design of these systems is the use of message-passing to allow objects (such as processes) to communicate. At a minimum, an object in a message-passing system must supply its kernel with a message and a destination identifier. The kernel is responsible for forwarding the message to the object that is associated with the destination identifier.[4] The message-passing paradigm has encouraged the study and development of algorithms for fault-tolerant, modular, and extensible applications, in both centralized and distributed systems.[5]

In general, most identifiers are formed as hierarchical structures consisting of a number of fields.[6] For example, many systems create their identifiers from two parts: a host-part that specifies the host on which the object resides, and a port-part that uniquely identifies the object on the host.[7] In this case, a message transmission requires the kernel to extract the host-part of the identifier to determine where the message should be routed (i.e. to the local host or a remote host). Should the message be destined for an object on a remote host, the host-part is mapped into a network-specific host-address. The identifier and the message are then copied into a packet for transmission over the network to the remote host.

An alternative to identifying objects with hierarchical identifiers is to use globally unique identifiers that contain no specific host or port references.[8] How the kernel handles globally unique identifiers is an implementation issue; however, two typical solutions are:[9]

0038–0644/93/040405–14$12.00
© 1993 by John Wiley & Sons, Ltd.

Received 24 June 1992
Revised 10 November 1992

Use of a commercial interface package

In the three years since the choices of software had to be made for this project, a number of commercial interface packages have appeared on the market, such as DataViews, from VI Corp., and SL-GMS from SL Corp. If the decisions as to software had to be made today, these packages would merit careful consideration. However, most commercial packages, regardless of their being labelled 'object-oriented', are not based on an object-oriented programming system and language. They are really libraries of routines written in a conventional high-level language. This means that the benefits of inheritance are lost: it is difficult or impossible to change the behaviour of the objects in the interface system. For our work, there is the additional problem that most such libraries presume a mouse-based style of user interaction; application of them to a strictly keyboard interface is not always supported.

Choosing a language today

Choices among object-oriented languages are wider today than three years ago. However, the same requirements still apply: the language must be fully object-oriented but must support easy access to the underlying services of the operating system. Typing, which was not present in Objective-C, would have caught few of the blunders we committed in development. More useful would be a robust, symbolic debugger and inspectors.

ACKNOWLEDGEMENT

The work described in this paper was sponsored by the U.S. Federal Aviation Administration under Contract No. DTFA01-84-C-00001.

REFERENCES

1. Peter Wegner, 'Dimensions of object-based language design', *OOPSLA '87 Proceedings*, Orlando, Florida, 4–6 October 1987. Association for Computing Machinery.
2. Glenn Krasner and Stephen Pope, 'A cookbook for using the model–view–controller user interface paradigm in Smalltalk-80', *Journal of Object-Oriented Programming*, **1**, (3), 26–49 (1988).
3. Kurt Schmucker, *Object-Oriented Programming for the Macintosh*, Hyden, Hasbrouck Heights, New Jersey, 1986.
4. Ralph Johnson and Brian Foote, 'Designing reusable classes', *Journal of Object-Oriented Programming*, **1**, (2), 22–35 (1988).
5. Adele Goldberg, *Smalltalk-80: the Interactive Programming Environment*, Addison Wesley, Reading, Massachusetts, 1984.
6. Brad Cox, *Object-Oriented Programming: an Evolutionary Approach*, Addison Wesley, Reading, Massachusetts, 1986.